AI DataLake

Flink Syntax Reference

Issue	01
Date	2026-11-06



HUAWEI TECHNOLOGIES CO., LTD.

Copyright © Huawei Technologies Co., Ltd. 2026. All rights reserved.

No part of this document may be reproduced or transmitted in any form or by any means without prior written consent of Huawei Technologies Co., Ltd.

Trademarks and Permissions


and other Huawei trademarks are trademarks of Huawei Technologies Co., Ltd.
All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between Huawei and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Huawei Technologies Co., Ltd.

Address: Huawei Industrial Base
Bantian, Longgang
Shenzhen 518129
People's Republic of China

Website: https://www.huawei.com

Email: support@huawei.com

Security Declaration

Vulnerability

Huawei's regulations on product vulnerability management are subject to the *Vul. Response Process.* For details about this process, visit the following web page:
https://www.huawei.com/en/psirt/vul-response-process
For vulnerability information, enterprise customers can visit the following web page:
https://securitybulletin.huawei.com/enterprise/en/security-advisory

Contents

1 Connectors....1
1.1 Supported Connectors....1
1.2 Kafka Connector....2
1.3 Doris Connector....14
1.4 Upsert-Kafka Connector....24
1.5 FileSystem Connector....33
1.6 DataGen Connector....42
1.7 BlackHole Connector....50
1.8 Print Connector....53
2 Keywords....56
3 DDL Statements....57
3.1 CREATE CATALOG....57
3.2 CREATE DATABASE....58
3.3 CREATE TABLE....60
3.4 CREATE VIEW....65
3.5 CREATE FUNCTION....67
3.6 ALTER Statement....68
3.7 DROP Statements....70
4 DML Statements....73
4.1 INSERT INTO....73
5 Queries Statements....77
6 Built-in Functions....79
6.1 Comparison Functions....79
6.2 Logical Functions....85
6.3 Arithmetic Functions....88
6.4 String Functions....95
6.5 Time Functions....105
6.6 Conditional Functions....113
6.7 Type Conversion Functions....117
6.8 Aggregate Functions....119
6.9 JSON Functions....126

 Copyright © Huawei Technologies Co., Ltd.

6.10 Value Construction Functions 131
6.11 Value Access Functions 133
6.12 Grouping Functions 134
6.13 Hash Functions 135
7 Auxiliary Statements 138
7.1 USE Statement 138
7.2 SET Statement 139
8 Enhanced Flink Features 142
8.1 Enhanced Functions 142
8.1.1 BITAND 142
8.1.2 BITNOT 143
8.1.3 BITOR 143
8.1.4 BITXOR 144
8.1.5 DATE_ADD 145
8.1.6 DATE_SUB 146
8.1.7 DATEDIFF 147
8.1.8 URL_DECODE 148
8.1.9 URL_ENCODE 149
8.2 Using RocksDB Merge Operator in FlinkSQL to Improve the Performance of the Join Operator 150
8.3 Using RocksDB Prefix Bloom Filter in FlinkSQL to Improve the Performance of the Join and Top N Operators 150
8.4 Optimizing the Performance of the SPLIT_INDEX Function in Flink SQL 151
8.5 Using Object Reuse of FlinkSQL to Improve Computing Performance 152
8.6 Flink SQL Lookup Join's Support for the PROCTIME() Syntax 153
8.7 Optimizing FlinkSQL JSON_VALUE Performance 154
8.8 Limiting Read Rate for Flink SQL Kafka and Upsert-Kafka Connector 154
8.9 Using Multi-Joins in FlinkSQL 155
8.10 Flink on Iceberg Supporting Dimension Table Join 157
8.11 Flink on Iceberg Supporting the VARIANT Data Type 159

 Copyright © Huawei Technologies Co., Ltd.

1 Connectors

1.1 Supported Connectors

Table 1-1 lists the supported connectors.

Flink supports two types of connectors: Iceberg and Paimon. For complete usage instructions for these two connectors (including preparations, data preparation, and other details), see **Iceberg on Flink** and **Paimon on Flink**. **Table 1-1** describes only the supported capabilities.

Table 1-1 Supported connectors

Connector	Source Table	Dimension Table	Result Table	Description
1.2 Kafka Connector	√	×	√	Reads from and writes data to DMS for Kafka message queues. Supports source tables and result tables (dimension tables not supported).
Iceberg Connector	√	√	√	Reads from and writes data to Iceberg lakehouse tables. Supports source tables, dimension tables, and result tables.
Paimon Connector	√	√	√	Reads from and writes data to Paimon lakehouse tables (unified stream and batch processing). Supports source tables, dimension tables, and result tables.

 Copyright © Huawei Technologies Co., Ltd.

Connector	Source Table	Dimension Table	Result Table	Description
1.3 Doris Connector	√	√	√	Reads from and writes data to CloudTable Doris databases. Supports efficient data writing via Stream Load. Supports source tables, dimension tables, and result tables.
1.4 Upsert-Kafka Connector	√	×	√	Reads from and writes data to DMS for Kafka with upsert semantics. Supports source tables and result tables (dimension tables not supported).
1.5 FileSystem Connector	√	×	√	Reads from and writes data to local file systems, OBS, and other storage. Supports unified batch and streaming read/write. Supports source tables and result tables (dimension tables not supported).
1.6 DataGen Connector	√	×	×	Generates random or sequential test data based on predefined rules. No external data source required. Supports source tables only.
1.7 BlackHole Connector	×	×	√	Discards all written data and produces no output. Supports result tables only.
1.8 Print Connector	×	×	√	Prints data to standard output for development and debugging purposes. Supports result tables only.

1.2 Kafka Connector

Overview

The Kafka connector is used to read data from or write data to **DMS for Kafka (Topic)**. It is the most commonly used streaming data connector and supports exactly-once semantics.

 Copyright © Huawei Technologies Co., Ltd.

Table 1-2 Connector details

Category	Details
Supported type	Source table, result table, and data ingestion target
Running mode	Stream mode
Data format	For details, see **Formats**.
API type	SQL, DataStream
Whether result table can be updated or deleted	Result table data cannot be updated or deleted, but can be inserted. For details about data update and deletion, see **1.4 Upsert-Kafka Connector**.

Application Scenarios

- **Real-time data ingestion**: Consumes messages from Kafka for stream processing and analysis.
- **Data pipeline**: Writes processed result data to Kafka for consumption by downstream systems.
- **Event-driven architecture**: Triggers real-time computing and responses based on Kafka messages.
- **Data integration**: Functions as a message broker between different systems to enable data transfer.
- **CDC data capture**: Works with the Debezium/Canal format to capture changed data.

Notes and Constraints

- Consumer offset committing behavior: If checkpointing is enabled, offsets are committed when the checkpoint completes. If checkpointing is disabled, offsets are automatically committed by the Kafka consumer.
- Topic must be created in advance for default data consumption.

Syntax Structure

```
CREATE TABLE {table_name} (
  {column_name} {column_type},
  ...
) WITH (
  'connector' = 'kafka',
  'topic' = '{kafka_topic}',
  'properties.bootstrap.servers' = '{broker_list}',
  'properties.group.id' = '{consumer_group}',
  'scan.startup.mode' = '{startup_mode}',
  'format' = '{data_format}',
  '{format_specific_param}' = '{value}'
);
```

 Copyright © Huawei Technologies Co., Ltd.

WITH Parameters

Source parameters

Parameter	Mandatory	Type	Default Value	Value Range	Description
connector	Yes	String	None	kafka	Specifies that the connector type is Kafka. This parameter is mandatory and must be set to the fixed value **kafka**. The Flink runtime loads the corresponding Kafka Source/Sink factory class based on this value and declares the table as a Kafka data source or output table.
topic	Yes	String	None	Valid topic name	Specifies the name of the Kafka topic to be consumed. This parameter is mandatory for the source. Multiple topics separated by semicolons (for example, **topic-a;topic-b**) are supported. The Flink runtime assigns a subtask to each topic partition for parallel consumption, and partitions from multiple topics are merged and scheduled together.
properties.bootstrap.servers	Yes	String	None	Host:port list	Specifies the list of Kafka broker addresses. Use commas (,) to separate multiple addresses. This parameter is mandatory for both the source and sink. The Flink runtime uses this configuration to create a Kafka Consumer or Producer client connection. You are advised to configure multiple broker addresses to ensure high availability.
properties.group.id	No	String	None	Valid consumer group name	Specifies the ID of a Kafka consumer group. This parameter is mandatory for the source. The Flink runtime uses this group ID to identify the consumer. Consumers with the same group ID share the consumption progress of topic partitions. Different jobs must use different group IDs to avoid partition allocation conflicts.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Type	Default Value	Value Range	Description
scan.startup.mode	No	String	group-offsets	earliest-offset/ latest-offset/ group-offsets/ specific-offsets/ timestamp	Specifies the start position mode when the source is consumed for the first time. The default value is **group-offsets**. ● If this parameter is set to **specific-offsets**, it must be used together with **scan.startup.specific-offsets**. ● When this parameter is set to **timestamp**, it must be used together with **scan.startup.timestamp-millis**. The Flink runtime determines the initial consumer offset of the Kafka consumer based on this mode, which affects the data read start point when the job starts.
scan.startup.specific-offsets	No	String	None	Partition:offset list	Specifies the start offset of each partition. This parameter takes effect only when **scan.startup.mode** is set to **specific-offsets**. The format is **partition:offset,partition:offset**. The Flink runtime sets the initial offset of each partition of the Kafka consumer to the specified position. For partitions that have not been consumed, data is read from the specified offset.
scan.startup.timestamp-millis	No	Long	None	Timestamp in milliseconds	Specifies the start timestamp of consumption, in milliseconds. This parameter takes effect only when **scan.startup.mode** is set to **timestamp**. The Flink runtime sets the starting offset of the Kafka consumer to the first message whose timestamp is not earlier than the specified timestamp. This is used to resume data consumption from a specific point in time.
format	Yes	String	None	json/csv/avro/ debezium-json/ canal-json/raw	Specifies the deserialization format of messages. This parameter is mandatory for both the source and sink. The Flink runtime loads the corresponding format factory class based on this value to convert between the Kafka message byte stream and Flink table row data.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Type	Default Value	Value Range	Description
scan.topic-partition-discovery.interval	No	Duration	5 min	Valid duration, for example, 30 seconds or 5 minutes.	Specifies the check interval for dynamic partition discovery. After this parameter is set, newly added partitions in a topic can be automatically discovered without restarting the job. The Flink runtime starts a background thread to periodically query the topic metadata at this interval. When a new partition is discovered, it is automatically assigned to an idle subtask for consumption.
properties.*	No	String	-	Kafka consumer configuration	Any configuration parameter passed to the Kafka consumer, prefixed with **properties.**. After removing the **properties.** prefix, the configuration item name is directly passed through to the Kafka consumer. The Flink runtime passes these parameters as-is when creating the Kafka client, which can be used to adjust consumption behavior (such as **fetch.min.bytes**, **max.poll.records**, etc.).

Sink parameters

Parameter	Mandatory	Type	Default Value	Value Range	Description
connector	Yes	String	None	kafka	Specifies that the connector type is Kafka. This parameter is mandatory and must be set to the fixed value **kafka**. The Flink runtime loads the corresponding Kafka Source/Sink factory class based on this value and declares the table as a Kafka data source or output table.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Type	Default Value	Value Range	Description
topic	Yes	String	None	Valid topic name	Specifies the name of the Kafka topic to which data is written. This parameter is mandatory for the sink. When Flink is running, all output records are sent to the topic. If the topic does not exist, you need to create the topic in advance or enable the function of automatically creating a topic in Kafka.
properties.bootstrap.servers	Yes	String	None	Host:port list	Specifies the list of Kafka broker addresses. Use commas (,) to separate multiple addresses. This parameter is mandatory for both the source and sink. The Flink runtime uses this configuration to create a Kafka Consumer or Producer client connection. You are advised to configure multiple broker addresses to ensure high availability.
format	Yes	String	None	json/csv/avro/debezium-json/canal-json/raw	Specifies the serialization format of messages. This parameter is mandatory for the sink. When Flink is running, it serializes the row data in the Flink table into byte streams in the corresponding format and writes the data to Kafka based on the value of this parameter.
sink.partitioner	No	String	default	default/fixed/round-robin/fully qualified name of a custom class	Specifies the partitioning policy for writing data to Kafka. The default value is **default**. The **default** policy partitions data by key hash (or uses round-robin if there is no key). The **fixed** policy writes data to a fixed partition for each sink subtask. The **round-robin** policy distributes data across all partitions in a round-robin manner. The Flink runtime determines the Kafka partition to which each record is sent based on this policy, affecting the data distribution balance across partitions.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Type	Default Value	Value Range	Description
sink.semantic	No	String	at-least-once	at-least-once/exactly-once/none	Specifies the semantic guarantee level for the write operation. The default value is **at-least-once**. If set to **exactly-once**, the Flink Checkpoint mechanism must be enabled. In **at-least-once** mode, the Flink runtime immediately commits each record. In **exactly-once** mode, Kafka transactions are used to commit data only after a checkpoint is complete, ensuring no duplicate data but increasing latency.
properties.*	No	String	-	Kafka producer configuration	Any configuration parameter passed to the Kafka producer, prefixed with **properties.**. After removing the **properties.** prefix, the configuration item name is directly passed through to the Kafka producer. The Flink runtime passes these parameters as-is when creating the Kafka producer, which can be used to adjust production behavior (such as **batch.size**, **linger.ms**, etc.).

Format parameters

The following are common format-related parameters. They take effect after the format is specified using the format parameter.

JSON format parameters

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Type	Default Value	Value Range	Description
json.fail-on-missing-field	No	Boolean	false	true/false	Specifies whether to report an error when a field defined in the DDL is missing in the JSON message. The default value is **false**. If set to **true**, an exception is thrown when a missing field is encountered, causing the job to fail. If set to **false**, the missing field is set to null and processing continues. The Flink runtime checks each field during deserialization. This parameter determines the fault tolerance policy when a field is missing.
json.ignore-parse-errors	No	Boolean	false	true/false	Specifies whether to ignore JSON parsing errors. The default value is **false**. If set to **true**, the field is skipped instead of reporting an error when a parsing error occurs. This improves fault tolerance but may cause data loss. When an exception occurs during deserialization, the **true** mode sets the abnormal field to null and continues processing subsequent fields. The **false** mode directly throws an exception to terminate the job.
json.timestamp-format.standard	No	String	SQL	SQL/ISO-8601	Specifies the parsing format standard for the timestamp type. The default value is **SQL**. The **SQL** format is parsed as **yyyy-MM-dd HH:mm:ss.S**, and the **ISO-8601** format is parsed as **yyyy-MM-dd'T'HH:mm:ss.S**. The Flink runtime selects the corresponding parser based on this standard, which affects the conversion rules between timestamp strings in JSON and the Flink TIMESTAMP type.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Type	Default Value	Value Range	Description
json.map-null-key.mode	No	String	FAIL	FAIL/DROP/LITERAL	Specifies how null keys in the Map type are handled. The default value is **FAIL**. **FAIL** throws an exception when a null key is encountered; **DROP** discards the key-value pair of the null key; **LITERAL** replaces the null key with a specified string. When serializing Map data, the Flink runtime processes null keys according to this policy, which affects the output behavior of Map data containing null keys.
json.map-null-key.literal	No	String	"null"	Any string	Specifies the replacement string for the null key when **json.map-null-key.mode** is set to **LITERAL**. This parameter takes effect only when **json.map-null-key.mode** is set to **LITERAL**. When the Flink runtime serializes Map data, it replaces the null key with the string value specified by this parameter for output.

CSV format parameters

Parameter	Mandatory	Type	Default Value	Value Range	Description
csv.field-delimiter	No	String	,	Single character	Specifies the delimiter of CSV fields. The default value is a comma (,). Only a single character is supported. The Flink runtime uses this delimiter to split or concatenate field values in CSV records.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Type	Default Value	Value Range	Description
csv.disable-quote-character	No	Boolean	false	true/false	Specifies whether to disable the quotation character. The default value is **false**. If this parameter is set to **true**, the quotation character is treated as a common character. When the Flink runtime parses CSV, field values that contain delimiters are no longer protected by quotation marks, which may cause field parsing errors.
csv.quote-character	No	String	"	Single character	Specifies the quotation character in the CSV file, which is used to enclose field values that contain delimiters. The default value is double quotation marks ("). This parameter takes effect only when **csv.disable-quote-character** is set to **false**. When the Flink runtime parses and generates CSV, it uses this character to identify and enclose field values that contain delimiters or newline characters.
csv.allow-comments	No	Boolean	false	true/false	Specifies whether the CSV file can contain comment lines (starting with **#**). The default value is **false**. If set to **true**, lines starting with **#** are skipped. When the Flink runtime parses CSV records, lines starting with **#** are treated as comments and ignored, and are not processed as data records.
csv.ignore-parse-errors	No	Boolean	false	true/false	Specifies whether to ignore CSV parsing errors. The default value is **false**. If set to **true**, fields that encounter parsing errors are set to null and processing continues. When the Flink runtime encounters a format exception during deserialization, in **true** mode, it skips the erroneous field and continues processing subsequent fields; in **false** mode, it directly throws an exception and terminates the job.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Type	Default Value	Value Range	Description
csv.array-element-delimiter	No	String	;	Any string	Specifies the delimiter for array elements. The default value is a semicolon (;). When the Flink runtime parses and generates array-type fields in CSV, this delimiter is used to split or concatenate array elements.

Avro format parameters

Parameter	Mandatory	Type	Default Value	Value Range	Description
encoding	No	String	binary	binary/json	Specifies the encoding format of Avro. **binary** indicates binary encoding, and **json** indicates JSON encoding. The Flink runtime selects the serialization/deserialization method for binary data based on this value. Binary encoding is more compact and efficient, while JSON encoding offers better readability.

Usage Example

- **Example 1: Consume JSON messages from a Kafka topic with the simplest configuration.**

```
-- Create a Kafka source table.
CREATE TABLE kafka_source (
  user_id BIGINT,
  event_type STRING,
  event_time TIMESTAMP(3)
) WITH (
  'connector' = 'kafka',
  'topic' = 'user-events',
  'properties.bootstrap.servers' = 'kafka-broker:9092',
  'properties.group.id' = 'flink-consumer-group',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json'
);

-- Create the Print result table.
CREATE TABLE print_sink (
  user_id BIGINT,
  event_type STRING,
  event_time TIMESTAMP(3)
) WITH (
```

 Copyright © Huawei Technologies Co., Ltd.

```
  'connector' = 'print'
);

-- Submit the job.
INSERT INTO print_sink
SELECT user_id, event_type, event_time
FROM kafka_source;
```

- **Example 2: Start consuming from the earliest offset of the Kafka topic. This is suitable for scenarios requiring full data replay.**

```
-- Create a Kafka source table (start consuming from the earliest offset).
CREATE TABLE kafka_source_earliest (
  order_id STRING,
  amount DECIMAL(10, 2),
  order_time TIMESTAMP(3)
) WITH (
  'connector' = 'kafka',
  'topic' = 'orders',
  'properties.bootstrap.servers' = 'kafka-broker:9092',
  'properties.group.id' = 'order-consumer',
  'scan.startup.mode' = 'earliest-offset',
  'format' = 'json'
);

-- Create the Print result table.
CREATE TABLE print_sink (
  order_id STRING,
  amount DECIMAL(10, 2),
  order_time TIMESTAMP(3)
) WITH (
  'connector' = 'print'
);

-- Submit the job.
INSERT INTO print_sink
SELECT order_id, amount, order_time
FROM kafka_source_earliest;
```

- **Example 3: Start consuming data from a specified timestamp. This is suitable for scenarios where data processing needs to be resumed from a specific point in time.**

```
-- Create a Kafka source table (start consuming from a specified timestamp).
CREATE TABLE kafka_source_timestamp (
  log_id STRING,
  log_level STRING,
  message STRING,
  log_time TIMESTAMP(3)
) WITH (
  'connector' = 'kafka',
  'topic' = 'application-logs',
  'properties.bootstrap.servers' = 'kafka-broker:9092',
  'properties.group.id' = 'log-consumer',
  'scan.startup.mode' = 'timestamp',
  'scan.startup.timestamp-millis' = '1704067200000',
  'format' = 'json'
);

-- Create the Print result table.
CREATE TABLE print_sink (
  log_id STRING,
  log_level STRING,
  message STRING,
  log_time TIMESTAMP(3)
) WITH (
  'connector' = 'print'
);

-- Submit the job.
INSERT INTO print_sink
```

 Copyright © Huawei Technologies Co., Ltd.

```
SELECT log_id, log_level, message, log_time
FROM kafka_source_timestamp;
```

- **Example 4: Consume Kafka messages in CSV format with a custom field delimiter.**

```
-- Create a Kafka source table in CSV format.
CREATE TABLE kafka_csv_source (
  id INT,
  name STRING,
  score DOUBLE
) WITH (
  'connector' = 'kafka',
  'topic' = 'csv-data',
  'properties.bootstrap.servers' = 'kafka-broker:9092',
  'properties.group.id' = 'csv-consumer',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'csv',
  'csv.field-delimiter' = '|'
);

-- Create the Print result table.
CREATE TABLE print_sink (
  id INT,
  name STRING,
  score DOUBLE
) WITH (
  'connector' = 'print'
);

-- Submit the job.
INSERT INTO print_sink
SELECT id, name, score
FROM kafka_csv_source;
```

1.3 Doris Connector

Overview

The Doris connector is used to read from and write data to the **CloudTable Doris** database, a table storage service. It supports efficient data writing via Stream Load and data reading through the Doris FE query interface.

Table 1-3 Connector details

Category	Details
Supported type	Source table, dimension table, and result table
Running mode	Stream mode and batch mode
Data format	Not supported currently
API type	SQL, DataStream
Whether result table can be updated or deleted	Yes

 Copyright © Huawei Technologies Co., Ltd.

Application Scenarios

- **Real-time data write**: Flink writes the real-time data processed by itself to the Doris OLAP table to support real-time report query.
- **Data synchronization**: Data from data sources such as Kafka is synchronized to Doris to build a real-time data warehouse.
- **OLAP analysis**: Dimension data is read from Doris and associated with real-time streams for analysis.
- **ETL output**: The Flink ETL processing result is written to Doris for BI tools to directly query.

Notes and Constraints

- To enable exactly-once write, you need to enable Flink Checkpoint and set a unique sink.label-prefix.
- When data is written to Doris, the field type must be compatible with the column type in the Doris table.

Syntax Structure

Source

```
CREATE TABLE {table_name} (
  {column_name} {column_type},
  ...
) WITH (
  'connector' = 'doris',
  'fenodes' = '{fe_host:http_port}',
  'table.identifier' = '{database}.{table}',
  'username' = '{username}',
  'password' = '{password}'
);
```

Sink

```
CREATE TABLE {table_name} (
  {column_name} {column_type},
  ...
) WITH (
  'connector' = 'doris',
  'fenodes' = '{fe_host:http_port}',
  'table.identifier' = '{database}.{table}',
  'username' = '{username}',
  'password' = '{password}',
  'sink.label-prefix' = '{label_prefix}',
  'sink.properties.format' = '{data_format}',
  'sink.enable-delete' = '{enable_delete}'
);
```

WITH Parameters

Common parameters

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Type	Default Value	Value Range	Description
connector	Yes	String	None	doris	Specifies that the connector type is Doris. This parameter is mandatory and must be set to the fixed value **doris**. The Flink runtime loads the Doris connector factory class based on this value and declares the table as a Doris data source or output table.
fenodes	Yes	String	None	Host:port list	Specifies the HTTP address of the Doris FE node. Use commas (,) to separate multiple addresses. This parameter is mandatory for both the source and sink. The Flink runtime uses this address to connect to the Doris FE node to obtain the query plan or submit a Stream Load request. You are advised to configure multiple FE addresses to implement high-availability failover.
table.identifier	Yes	String	None	database.table	Specifies the Doris table identifier in the format of database_name.table_name. This parameter is mandatory for both the source and sink. The Flink runtime uses this identifier to locate the target table in Doris, which determines the data read and write targets.
username	Yes	String	None	Valid username	Specifies the username of the Doris database. This parameter is mandatory for both the source and sink. The Flink runtime uses this username to authenticate with Doris. You must have the read and write permissions on the corresponding table.
password	Yes	String	None	Valid password	Specifies the password of the Doris database. This parameter is mandatory for both the source and sink. The Flink runtime uses this password together with the username to authenticate with Doris.

Source parameters

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Type	Default Value	Value Range	Description
doris.read.field	No	String	None	Column name list	Specifies the columns to be read. Multiple columns are separated by commas (,). If this parameter is not specified, all columns are read. The Flink runtime pushes down this column list to the SELECT clause of the Doris query SQL statement, reducing unnecessary data transmission and improving read efficiency.
doris.filter.query	No	String	None	SQL filter condition	Specifies the filter condition for reading data, which is pushed down to Doris for execution. When Flink is running, this condition is appended to the WHERE clause of the Doris query SQL statement to filter data on the Doris side, reducing the amount of data transmitted to Flink.
doris.request.tablet.size	No	Integer	2	Positive integer	Specifies the number of tablets that can be concurrently read from each Doris BE partition. When Flink is running, tablets are allocated to multiple read subtasks based on this value for parallel execution. Increasing this value can improve the read parallelism but increase the load on the Doris side.
doris.request.connect.timeout.ms	No	Integer	30000	Positive integer	Specifies the timeout interval for connecting to the Doris BE, in milliseconds. When Flink runtime establishes a connection to the Doris BE, a timeout exception is thrown if the connection is not successfully established within this time. You can increase this value if the network is unstable.
doris.request.read.timeout.ms	No	Integer	30000	Positive integer	Specifies the timeout interval for reading data from the Doris BE, in milliseconds. When Flink runtime reads data from the Doris BE, a timeout exception is thrown if no response is received within this time. You can increase this value in large query scenarios.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Type	Default Value	Value Range	Description
doris.request.query.timeout.s	No	Integer	3600	Positive integer	Specifies the timeout interval for querying Doris, in seconds. After the Flink runtime submits a query to Doris, if the query is not completed within the specified time, the query is canceled. You can increase this value in full read scenarios for large tables.
doris.request.retries	No	Integer	3	Positive integer	Specifies the number of retry times for requesting Doris. When a request fails, the Flink runtime automatically retries. If the number of retry times reaches the maximum, an exception is thrown. You can increase this value if the network is unstable.
doris.deserialize.arrow.async	No	Boolean	false	true/false	Specifies whether to deserialize Arrow data asynchronously. The default value is false, indicating synchronous deserialization. If this parameter is set to **true**, the Flink runtime uses an independent thread pool to asynchronously convert Arrow to RowData. This increases read throughput but also increases memory consumption.
doris.deserialize.queue.size	No	Integer	64	Positive integer	Specifies the size of the asynchronous deserialization queue. This parameter takes effect only when **doris.deserialize.arrow.async** is set to **true**. In asynchronous deserialization mode, the Flink runtime uses this queue to buffer deserialized data. When the queue is full, the deserialization thread is blocked and waits.

Sink parameters

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Type	Default Value	Value Range	Description
sink.label-prefix	No	String	None	Valid label prefix	Specifies the label prefix for Stream Load. You are advised to set a unique prefix for each job to ensure exactly-once semantics. The Flink runtime generates a Stream Load label using the label prefix and a unique identifier. Doris uses the label for deduplication to ensure that the same batch of data is not written repeatedly.
sink.properties.format	No	String	json	json / csv	Specifies the data transfer format for Stream Load. The default value is **json**. The Flink runtime serializes Flink row data into the specified format and sends it to the Doris Stream Load API. The **json** format offers good compatibility, while the **csv** format results in a smaller data volume.
sink.properties.max_filter_ratio	No	String	0	0–1	Specifies the maximum filtering ratio of Stream Load. If the filtering ratio of a batch of data exceeds this value, the entire batch fails to be written. After Stream Load is complete, the Flink runtime checks the filtering ratio. If the ratio exceeds the threshold, an exception is thrown and the batch is retried. If this parameter is set to **0**, no filtering error is allowed.
sink.enable-delete	No	Boolean	true	true/false	Controls whether to enable the delete operation. The default value is **false**. After this parameter is enabled, the retract records (-U and -D) of Flink are converted into the DELETE operation of Doris. When Flink serializes records, it generates Doris delete markers for retract records instead of common writes. Only the Doris Unique Key model table supports the delete operation.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Type	Default Value	Value Range	Description
sink.buffer-size	No	MemorySize	1 MB	Valid memory size, for example, 128 MB or 256 MB.	Sets the size of the sink buffer. Flink buffers data in memory during runtime. When the buffer is full, a Stream Load submission is triggered. Increasing this value reduces the number of submissions and improves throughput, but it also increases memory usage and data latency.
sink.buffer.count	No	Integer	3	Positive integer	Sets the number of sink buffers. Flink maintains multiple buffers and uses them in turn during runtime. When one buffer is being submitted, other buffers continue to receive data. Increasing this value improves write concurrency, but it also increases memory usage.
sink.max-retries	No	Integer	1	Positive integer	Specifies the maximum number of retry attempts for failed writes. When a Stream Load submission fails during Flink runtime, the system automatically retries. The job fails if the maximum number of retries is exhausted. You can increase this value when the network or Doris is unstable.
sink.check-interval	No	Duration	0s	Valid duration, for example, 30 seconds or 5 minutes.	Specifies the interval for checking whether the buffer needs to be flushed, in milliseconds. During Flink runtime, a background thread is started to check the buffer status at this interval. When the flush conditions are met, a Stream Load submission is triggered. A shorter interval results in lower data latency but higher check overhead.

Lookup parameters

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Type	Default Value	Value Range	Description
lookup.cache.max-rows	No	Long	-1	Positive integer	Specifies the maximum number of rows in the dimension table lookup cache. This parameter is valid only in the Lookup JOIN scenario. The Flink runtime maintains an LRU cache in memory. When the number of cached rows reaches this upper limit, the least recently accessed record is evicted. Increasing this value improves the query hit ratio but increases memory consumption.
lookup.cache.ttl	No	Duration	10s	Positive integer + time unit	Specifies the time to live (TTL) for the dimension table lookup cache. This parameter is valid only in the Lookup JOIN scenario. When the TTL of cached records expires, Flink invalidates the records and loads them from Doris again during the next access. This balances data timeliness and query performance.
lookup.max-retries	No	Integer	1	Positive integer	Specifies the maximum number of retries for dimension table query. When Flink fails to query Doris, it automatically retries. If the number of retries reaches the maximum, an exception is thrown. You can increase the value of this parameter if the Doris load is high.
jdbc-url	Yes	String	None	None	Specifies the MySQL protocol address for connecting to Doris.

Usage Example

- **Example 1: Read data from the Doris table in batches.**

```
-- Create a Doris source table.
CREATE TABLE doris_source (
  user_id BIGINT,
  user_name STRING,
  age INT,
  city STRING
) WITH (
  'connector' = 'doris',
  'fenodes' = 'doris-fe:8030',
  'table.identifier' = 'user_db.user_info',
  'username' = 'root',
  'password' = 'xxxx'
);

-- Create the Print result table.
```

 Copyright © Huawei Technologies Co., Ltd.

```
CREATE TABLE print_sink (
  user_id BIGINT,
  user_name STRING,
  age INT,
  city STRING
) WITH (
  'connector' = 'print',
  'print-identifier' = 'doris-data'
);

-- Submit the job.
INSERT INTO print_sink
SELECT user_id, user_name, age, city
FROM doris_source;
```

- **Example 2: Write data to the Doris table in JSON format.**

```
-- Create a DataGen source table.
CREATE TABLE datagen_source (
  user_id BIGINT,
  event_type STRING,
  event_time TIMESTAMP(3),
  event_count BIGINT
) WITH (
  'connector' = 'datagen',
  'rows-per-second' = '100',
  'fields.user_id.kind' = 'sequence',
  'fields.user_id.start' = '1',
  'fields.user_id.end' = '100000',
  'fields.event_count.kind' = 'sequence',
  'fields.event_count.start' = '1',
  'fields.event_count.end' = '100'
);

-- Create the Doris result table.
CREATE TABLE doris_sink (
  user_id BIGINT,
  event_type STRING,
  event_time TIMESTAMP(3),
  event_count BIGINT
) WITH (
  'connector' = 'doris',
  'fenodes' = 'doris-fe:8030',
  'table.identifier' = 'analytics_db.user_events',
  'username' = 'root',
  'password' = 'xxxx',
  'sink.label-prefix' = 'flink-user-events',
  'sink.properties.format' = 'json',
  'sink.max-retries' = '3'
);

-- Submit the job.
INSERT INTO doris_sink
SELECT user_id, event_type, event_time, event_count
FROM datagen_source;
```

- **Example 3: Consume real-time event data from Kafka, perform aggregation, and write the results to Doris to support real-time OLAP queries.**

```
-- Kafka source table
CREATE TABLE kafka_event_source (
  user_id BIGINT,
  event_type STRING,
  page_url STRING,
  event_time TIMESTAMP(3),
  WATERMARK FOR event_time AS event_time - INTERVAL '5' SECOND
) WITH (
  'connector' = 'kafka',
  'topic' = 'user-events',
  'properties.bootstrap.servers' = 'kafka-broker:9092',
```

 Copyright © Huawei Technologies Co., Ltd.

```
  'properties.group.id' = 'doris-writer',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json',
  'json.ignore-parse-errors' = 'true'
);

-- Doris result table
CREATE TABLE doris_event_sink (
  window_start TIMESTAMP(3),
  window_end TIMESTAMP(3),
  event_type STRING,
  event_count BIGINT,
  unique_users BIGINT
) WITH (
  'connector' = 'doris',
  'fenodes' = 'doris-fe:8030',
  'table.identifier' = 'analytics_db.event_stats',
  'username' = 'root',
  'password' = 'xxxx',
  'sink.label-prefix' = 'flink-event-stats',
  'sink.properties.format' = 'json',
  'sink.enable-delete' = 'false'
);

-- 5-minute window aggregation and write
INSERT INTO doris_event_sink
SELECT
  window_start AS window_start,
  window_end AS window_end,
  event_type,
  COUNT(*) AS event_count,
  COUNT(DISTINCT user_id) AS unique_users
FROM TABLE(
  HOP(TABLE kafka_event_source, DESCRIPTOR(event_time), INTERVAL '1' MINUTE, INTERVAL '5'
MINUTE)
)
GROUP BY window_start, window_end, event_type;
```

- **Example 4: Use a Doris table as a dimension table and associate it with Kafka real-time streams through Lookup JOIN.**

```
-- Doris dimension table
CREATE TABLE doris_dim (
  user_id BIGINT,
  user_name STRING,
  city STRING,
  PRIMARY KEY (user_id) NOT ENFORCED
) WITH (
  'connector' = 'doris',
  'fenodes' = 'doris-fe:8030',
  'table.identifier' = 'user_db.user_info',
  'jdbc-url' = 'jdbc:mysql://<FE-instance-IP-address>:<FE-query-connection-port>',
  'username' = 'root',
  'password' = 'xxxx',
  'lookup.cache.max-rows' = '10000',
  'lookup.cache.ttl' = '10min'
);

-- Kafka real-time stream
CREATE TABLE kafka_realtime_source (
  user_id BIGINT,
  event_type STRING,
  event_time TIMESTAMP(3),
  proctime AS PROCTIME()
) WITH (
  'connector' = 'kafka',
  'topic' = 'events',
  'properties.bootstrap.servers' = 'kafka-broker:9092',
  'properties.group.id' = 'doris-lookup',
  'scan.startup.mode' = 'earliest-offset',
  'format' = 'json'
```

 Copyright © Huawei Technologies Co., Ltd.

```
);

-- Create the Print result table.
CREATE TABLE print_sink (
  user_id BIGINT,
  user_name STRING,
  city STRING,
  event_type STRING,
  event_time TIMESTAMP(3)
) WITH (
  'connector' = 'print',
  'print-identifier' = 'doris-lookup-result'
);

-- Submit the job: Output after Lookup JOIN.
INSERT INTO print_sink
SELECT e.user_id, d.user_name, d.city, e.event_type, e.event_time
FROM kafka_realtime_source e
LEFT JOIN doris_dim FOR SYSTEM_TIME AS OF e.proctime AS d
ON e.user_id = d.user_id;
```

- **Example 5: Enable the delete capability of the Doris sink to convert Flink retraction records into Doris delete operations.**

```
-- Create a DataGen source table.
CREATE TABLE datagen_source (
  user_id BIGINT,
  user_name STRING,
  city STRING
) WITH (
  'connector' = 'datagen',
  'rows-per-second' = '100',
  'fields.user_id.kind' = 'sequence',
  'fields.user_id.start' = '1',
  'fields.user_id.end' = '10000',
  'fields.user_name.length' = '10'
);

-- Create the Doris Upsert result table.
CREATE TABLE doris_upsert_sink (
  user_id BIGINT,
  user_name STRING,
  city STRING,
  PRIMARY KEY (user_id) NOT ENFORCED
) WITH (
  'connector' = 'doris',
  'fenodes' = 'doris-fe:8030',
  'table.identifier' = 'user_db.user_info',
  'username' = 'root',
  'password' = 'xxxx',
  'sink.label-prefix' = 'flink-user-upsert',
  'sink.properties.format' = 'json',
  'sink.enable-delete' = 'true'
);

-- Submit the job.
INSERT INTO doris_upsert_sink
SELECT user_id, user_name, city
FROM datagen_source;
```

1.4 Upsert-Kafka Connector

Overview

The Upsert-Kafka connector is a changelog variant of the Kafka connector, used to read and write **DMS for Kafka topics** in upsert mode. Unlike the regular Kafka connector, which only processes append data, the Upsert-Kafka connector

 Copyright © Huawei Technologies Co., Ltd.

interprets Kafka messages as changelog records, supporting insert, update, and delete operations by primary key. It is suitable for change data capture (CDC) scenarios that require end-to-end exactly-once semantics.

Category	Details
Supported type	Source table and result table
Running mode	Stream mode
Data format	For details, see **Formats**.
API type	SQL, DataStream
Whether result table can be updated or deleted	Yes

Application Scenarios

- **CDC data pipeline**: Writes change data captured by CDC tools such as Debezium to Kafka in upsert mode. When the downstream system consumes the data, it is automatically merged into the latest state based on the primary key.
- **Real-time data warehouse synchronization**: Writes retraction records (-U/+U) generated by Flink aggregation calculations to Kafka, allowing downstream consumers to obtain complete change semantics.
- **Cross-system data synchronization**: Serves as a change data transmission channel between different systems to ensure data consistency.
- **Event sourcing**: Persists domain events to Kafka in the form of change logs, supporting state reconstruction and time-travel queries.
- **Dimension table broadcasting**: Broadcasts change data of dimension tables to Kafka, and downstream jobs update the local dimension cache based on the primary key.

Notes and Constraints

- It is incompatible with the common Kafka connector (connector = 'kafka'). You are not advised to use both connectors to read and write the same topic.
- Topic must be created in advance for default data consumption.

 Copyright © Huawei Technologies Co., Ltd.

Differences from the Common Kafka Connector

Item	Kafka Connector	Upsert-Kafka Connector
Connector value	kafka	upsert-kafka
Primary key requirement	Not required	**PRIMARY KEY** must be defined.
Data semantics	Append-only stream	Change log stream (Changelog)
Source output	Original message	Latest status after deduplication by primary key
Sink write	Append-only write	Upsert write by primary key (update or insert)
Deletion support	Not supported	Sink supports deletion (the -D record is written as **NULL**).
Application scenario	Event stream and log collection	CDC synchronization, status update, and dimension broadcast
Message format	Common key-value pair	The key is the primary key field value, and the value is a complete record or **NULL** (deletion).

Syntax Structure

```
CREATE TABLE {table_name} (
  {column_name} {column_type},
  ...
  PRIMARY KEY ({pk_column}) NOT ENFORCED,
  WATERMARK FOR {time_column} AS {watermark_expression}
) WITH (
  'connector' = 'upsert-kafka',
  'topic' = '{kafka_topic}',
  'properties.bootstrap.servers' = '{broker_list}',
  'properties.group.id' = '{consumer_group}',
  'key.format' = '{key_format}',
  'value.format' = '{value_format}'
);
```

NOTE

The Upsert-Kafka connector must specify the serialization formats for both the key and the value. The key corresponds to the primary key field, and the value corresponds to the non-primary key fields. When a record is deleted, the value is **NULL**.

WITH Parameters

Source parameters

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Type	Default Value	Value Range	Description
connector	Yes	String	None	upsert-kafka	Specifies that the connector type is Upsert-Kafka. This parameter is mandatory and must be set to the fixed value **upsert-kafka**. The Flink runtime loads the Upsert-Kafka source factory class based on this value, declares the table as a changelog data source, and deduplicates and merges messages by primary key.
topic	Yes	String	None	Valid topic name	Specifies the name of the Kafka topic to be consumed. This parameter is mandatory for the source. The Flink runtime allocates a subtask to each topic partition for parallel consumption. Only the latest value is retained for records with the same primary key.
properties.bootstrap.servers	Yes	String	None	Host:port list	Specifies the list of Kafka broker addresses. Use commas (,) to separate multiple addresses. This parameter is mandatory for both the source and sink. The Flink runtime uses this configuration to create a Kafka Consumer or Producer client connection. You are advised to configure multiple broker addresses to ensure high availability.
properties.group.id	No	String	None	Valid consumer group name	Specifies the ID of a Kafka consumer group. This parameter is mandatory for the source. Flink uses this group ID to identify the consumer during running. Consumers with the same group ID share the consumption offset of topic partitions.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Type	Default Value	Value Range	Description
key.format	Yes	String	None	json/csv/avro	Specifies the deserialization format of the primary key field. This parameter is mandatory for both the source and sink. Flink deserializes the key part (primary key field) of a Kafka message based on this format during running. The serialization and deserialization of the primary key field must use the same format as that used for writing.
value.format	Yes	String	None	json/csv/avro	Specifies the deserialization format of non-primary key fields. This parameter is mandatory for both the source and sink. The Flink runtime deserializes the value part (non-primary key fields) of Kafka messages based on this format. The value of a deleted record is **NULL**.
scan.topic-partition-discovery.interval	No	Duration	None	Positive integer (ms)	Specifies the check interval for dynamic partition discovery. After this parameter is set, new partitions in the topic can be automatically discovered.
properties.*	No	String	-	Kafka consumer configuration	Any configuration parameter passed to the Kafka consumer, prefixed with **properties.**. After removing the **properties.** prefix, the configuration item name is directly passed through to the Kafka consumer.

Sink parameters

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Type	Default Value	Value Range	Description
connector	Yes	String	None	upsert-kafka	Specifies that the connector type is Upsert-Kafka. This parameter is mandatory and must be set to the fixed value **upsert-kafka**. When Flink is running, it loads the Upsert-Kafka Sink factory class based on the value of this parameter, declares the table as a changelog output table, and performs upsert write based on the primary key.
topic	Yes	String	None	Valid topic name	Specifies the name of the Kafka topic to which data is written. This parameter is mandatory for the sink. When Flink is running, it sends all output records to this topic. The key is the primary key field value, and the value is a complete record or **NULL** (indicating that the record is deleted).
properties.bootstrap.servers	Yes	String	None	Host:port list	Specifies the list of Kafka broker addresses. Use commas (,) to separate multiple addresses. This parameter is mandatory for both the source and sink.
key.format	Yes	String	None	json/csv/avro	Specifies the serialization format of the primary key field. This parameter is mandatory for the sink. The Flink runtime serializes the primary key field into the Key part of the Kafka message based on this format, which is used for partition routing and primary key identification.
value.format	Yes	String	None	json/csv/avro	Specifies the serialization format of the non-primary key field. This parameter is mandatory for the sink. The Flink runtime serializes non-primary key fields into the value part of Kafka messages based on this format. When a record is deleted, the value is serialized to **NULL**.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Type	Default Value	Value Range	Description
sink.buffer-flush.max-rows	No	Integer	None	Positive integer	Specifies the maximum number of rows in the sink buffer. When the buffer is full, batch writes are triggered. Increasing this value reduces the number of writes and improves throughput, but increases latency. This parameter is valid only for **at-least-once** and **none** semantics.
sink.buffer-flush.interval	No	Duration	None	Valid duration, for example, 10 seconds or 1 minute.	Specifies the interval for flushing the sink buffer. Batch writes are triggered at this interval. This parameter and **sink.buffer-flush.max-rows** together control the flush policy. The flush is triggered when either condition is met. This parameter is valid only for **at-least-once** and **none** semantics.
properties.*	No	String	-	Kafka producer configuration	Any configuration parameter passed to the Kafka producer, prefixed with **properties.**. After removing the **properties.** prefix, the configuration item name is directly passed through to the Kafka producer.

Key and value format parameters

The Upsert-Kafka connector requires specifying the formats for both the key and the value. Each format has its own independent parameters, with the prefixes **key.** and **value.**, respectively.

JSON format parameters (Using value as an example; replace the prefix with key. in the same way.)

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Type	Default Value	Value Range	Description
value.json.fail-on-missing-field	No	Boolean	false	true/false	Specifies whether to report an error when a field defined in the DDL is missing in the JSON message. The default value is **false**. If set to **true**, an exception is thrown when a missing field is encountered. If set to **false**, the missing field is set to null and processing continues.
value.json.ignore-parse-errors	No	Boolean	false	true/false	Specifies whether to ignore JSON parsing errors. The default value is **false**. If set to **true**, the field is skipped instead of reporting an error when a parsing error occurs. This improves fault tolerance but may cause data loss.
value.json.timestamp-format.standard	No	String	SQL	SQL/ISO-8601	Specifies the parsing format standard for the timestamp type. The default value is **SQL**. The **SQL** format is parsed as **yyyy-MM-dd HH:mm:ss.S**, and the **ISO-8601** format is parsed as **yyyy-MM-dd'T'HH:mm:ss.S**.

CSV format parameters (using value as an example)

Parameter	Mandatory	Type	Default Value	Value Range	Description
value.csv.field-delimiter	No	String	,	Single character	Specifies the delimiter of CSV fields. The default value is a comma (,).
value.csv.ignore-parse-errors	No	Boolean	false	true/false	Specifies whether to ignore CSV parsing errors. The default value is **false**.

 Copyright © Huawei Technologies Co., Ltd.

Usage Example

- **Example 1: Consume changelog data from the Upsert-Kafka topic, automatically deduplicate and merge the data based on the primary key, and output the data to the Print result table.**

```
-- Create an Upsert-Kafka source table in JSON format.
CREATE TABLE upsert_kafka_order_source (
  order_id STRING,
  user_id BIGINT,
  amount DECIMAL(10, 2),
  status STRING,
  update_time TIMESTAMP(3),
  PRIMARY KEY (order_id) NOT ENFORCED
) WITH (
  'connector' = 'upsert-kafka',
  'topic' = 'order-changes',
  'properties.bootstrap.servers' = 'kafka-broker:9092',
  'properties.group.id' = 'order-consumer',
  'key.format' = 'json',
  'value.format' = 'json'
);

-- Create the Print result table.
CREATE TABLE print_sink (
  order_id STRING,
  user_id BIGINT,
  amount DECIMAL(10, 2),
  status STRING,
  update_time TIMESTAMP(3)
) WITH (
  'connector' = 'print'
);

-- Submit the job.
INSERT INTO print_sink
SELECT order_id, user_id, amount, status, update_time
FROM upsert_kafka_order_source;
```

- **Example 2: The key is in CSV format and the value is in JSON format. This format is applicable to scenarios where there are a small number of primary key fields.**

```
-- Create an Upsert-Kafka source table (Key=CSV, Value=JSON).
CREATE TABLE upsert_kafka_mixed_source (
  user_id BIGINT,
  user_name STRING,
  city STRING,
  member_level STRING,
  update_time TIMESTAMP(3),
  PRIMARY KEY (user_id) NOT ENFORCED
) WITH (
  'connector' = 'upsert-kafka',
  'topic' = 'user-profile-changes',
  'properties.bootstrap.servers' = 'kafka-broker:9092',
  'properties.group.id' = 'profile-consumer',
  'key.format' = 'csv',
  'value.format' = 'json'
);

-- Create the Print result table.
CREATE TABLE print_sink (
  user_id BIGINT,
  user_name STRING,
  city STRING,
  member_level STRING,
  update_time TIMESTAMP(3)
) WITH (
  'connector' = 'print'
);
```

 Copyright © Huawei Technologies Co., Ltd.

```
-- Submit the job.
INSERT INTO print_sink
SELECT user_id, user_name, city, member_level, update_time
FROM upsert_kafka_mixed_source;
```

- **Example 3: The Upsert-Kafka end-to-end CDC pipeline forwards change data from one topic to another.**

```
-- Upstream CDC source table
CREATE TABLE upsert_kafka_cdc_source (
  order_id STRING,
  user_id BIGINT,
  amount DECIMAL(10, 2),
  status STRING,
  PRIMARY KEY (order_id) NOT ENFORCED
) WITH (
  'connector' = 'upsert-kafka',
  'topic' = 'cdc-order-changes',
  'properties.bootstrap.servers' = 'kafka-broker:9092',
  'properties.group.id' = 'cdc-relay-consumer',
  'key.format' = 'json',
  'value.format' = 'json'
);

-- Downstream forwarding target table
CREATE TABLE upsert_kafka_cdc_target (
  order_id STRING,
  user_id BIGINT,
  amount DECIMAL(10, 2),
  status STRING,
  PRIMARY KEY (order_id) NOT ENFORCED
) WITH (
  'connector' = 'upsert-kafka',
  'topic' = 'cdc-order-relay',
  'properties.bootstrap.servers' = 'kafka-broker:9092',
  'key.format' = 'json',
  'value.format' = 'json'
);

-- Forward change logs.
INSERT INTO upsert_kafka_cdc_target
SELECT order_id, user_id, amount, status
FROM upsert_kafka_cdc_source;
```

1.5 FileSystem Connector

Overview

The FileSystem connector is used to exchange data with the file system in Flink jobs of AI DataLake. This connector supports unified batch and stream read and write operations on storage backends such as the **local file system** and **OBS**. It provides multi-file format and partitioned write capabilities, and is applicable to scenarios such as data archiving, file export, and batch processing ETL.

Table 1-4 Connector details

Category	Details
Supported type	Source table and result table

 Copyright © Huawei Technologies Co., Ltd.

Category	Details
Data format	CSV, JSON, Avro, Parquet, Orc, Debezium-JSON, Canal-JSON, Raw
API type	SQL, DataStream
Whether result table can be updated or deleted	No

Application Scenarios

- **Data archiving**: Stream data is written to HDFS or OBS in time-partitioned rolling files for long-term archival storage.
- **File export**: Processing results are exported to the file system in CSV or JSON format for consumption by downstream systems.
- **Batch processing ETL**: Data is read from the file system in batches, cleansed, transformed, and then written to the target storage.
- **Columnar storage optimization**: Data is written to files in Parquet or ORC format to improve downstream query performance.
- **Multi-source data aggregation**: Data from multiple sources is written to the file system in a unified manner to build a data lake.

Notes and Constraints

- The file format must match the configuration of the format parameter. Otherwise, an error is reported during reading.
- Exactly-once semantics depends on the checkpoint mechanism. Ensure that checkpoints are triggered properly and at a reasonable interval.

Syntax Structure

Source

```
CREATE TABLE fs_source (
  {column_name} {column_type},
  ...
) WITH (
  'connector' = 'filesystem',
  'path' = '{file_system_path}',
  'format' = '{file_format}'
);
```

Sink

```
CREATE TABLE fs_sink (
  {column_name} {column_type},
  ...
) PARTITIONED BY ({partition_columns}) WITH (
  'connector' = 'filesystem',
  'path' = '{file_system_path}',
  'format' = '{file_format}',
  'sink.rolling-policy.file-size' = '{size}',
```

 Copyright © Huawei Technologies Co., Ltd.

```
  'sink.rolling-policy.rollover-interval' = '{interval}'
);
```

WITH Parameters

Common parameters

Parameter	Mandatory	Type	Default Value	Value Range	Description
connector	Yes	String	None	filesystem	Specifies the connector type as FileSystem. This parameter has a fixed value and cannot be modified. You must set this parameter to **filesystem**. The system loads the corresponding file system connector implementation based on the value of this parameter.
path	Yes	String	None	Valid file system path	Specifies the read and write path of the file system. Local paths (file:///), OBS paths (obs://), and other paths are supported. Ensure that the Flink job has read and write permissions on the path.
format	Yes	String	None	csv, json, avro, parquet, orc	Specifies the serialization format of the file. ● **csv**: Comma-separated text format. ● **json**: JSON line format. ● **avro**: Apache Avro line format. ● **parquet**: Columnar storage format. ● **orc**: Optimized Row Columnar format. You need to select a format based on data features and query requirements.

Source-specific parameters

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Type	Default Value	Value Range	Description
source.monitor-interval	No	Duration	None	Valid duration, for example, 10 seconds or 1 minute.	Specifies the interval for monitoring new files in the directory in streaming mode. After this parameter is configured, the source periodically scans the directory and automatically reads new files. If this parameter is not configured, the source reads all files in the directory only once. You need to adjust the scanning interval based on the file generation frequency.

Sink-specific parameters

Parameter	Mandatory	Type	Default Value	Value Range	Description
sink.rolling-policy.file-size	No	MemorySize	128 MB	Valid memory size, for example, 64 MB or 256 MB.	Specifies the target size of a rolling file. When a single file reaches this size, a new file is generated. You need to balance the number of files and the size of a single file. If the file size is too small, too many small files are generated, affecting query performance. If the file size is too large, write latency is affected.
sink.rolling-policy.rollover-interval	No	Duration	30min	Valid duration, for example, 30 seconds or 5 minutes.	Specifies the interval for rolling files. When the file opening time exceeds this interval, a new file is generated. You need to adjust the interval based on data visibility requirements. If the interval is too short, too many small files are generated. If the interval is too long, data latency is high.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Type	Default Value	Value Range	Description
sink.rolling-policy.check-interval	No	Duration	60s	Valid duration, for example, 30 seconds or 5 minutes.	Check interval of the time-based rolling policy. This parameter controls the frequency at which the system checks whether some part files should be rolled based on the **sink.rolling-policy.rollover-interval** parameter.
sink.partition-commit.policy.kind	No	String	None	success-file, metadata, success-file+metadata	Specifies the partition commit policy. • **success-file** indicates that the **_SUCCESS** marker file is created during the commit. • **metadata** indicates that partition metadata is written during the commit. After this parameter is configured, the system triggers the commit operation after the partition data is written, and notifies the downstream that the partition is ready.
sink.partition-commit.delay	No	Duration	0s	Valid duration, for example, 1 minute or 10 minutes.	Specifies the delay time for committing a partition. After the partition data is written, the system waits for the specified time before committing the data. This ensures that the delayed data is written to the correct partition. Set this parameter based on the data delay. If the delay is too short, data may be written to an incorrect partition.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Type	Default Value	Value Range	Description
sink.partition-commit.trigger	No	String	process-time	process-time, partition-time	Specifies the trigger type for committing a partition. • **process-time** indicates that the commit is triggered based on the system processing time. • **partition-time** indicates that the commit is triggered based on the partition time (watermark). You are advised to use **partition-time** to ensure data consistency.
auto-compaction	No	Boolean	false	true, false	Specifies whether to enable automatic file merging. After this function is enabled, the system automatically merges small files into files of the target size during checkpoint submission. You are advised to enable this function in scenarios where a large number of small files are generated due to a short rolling interval.
compaction.file-size	No	MemorySize	None	Valid memory size, for example, 128 MB or 256 MB.	Specifies the size of the target file after file merging. This parameter takes effect only when **auto-compaction** is set to **true**. It must be configured in coordination with **sink.rolling-policy.file-size**. You are advised to set this parameter to a value that is one to two times the value of **file-size**.

Usage Example

- **Example 1: Read CSV files from HDFS in batches and output the data to the Print result table.**

```
-- Create a FileSystem source table (CSV format, batch read).
CREATE TABLE fs_csv_source (
  order_id BIGINT,
  user_id BIGINT,
  amount DOUBLE,
  order_time TIMESTAMP(3)
) WITH (
  'connector' = 'filesystem',
  'path' = 'obs://xxx/data/orders/',
```

 Copyright © Huawei Technologies Co., Ltd.

```
  'format' = 'csv'
);

-- Create the Print result table.
CREATE TABLE print_sink (
  order_id BIGINT,
  user_id BIGINT,
  amount DOUBLE,
  order_time TIMESTAMP(3)
) WITH (
  'connector' = 'print',
  'print-identifier' = 'fs_csv_orders'
);

-- Submit the job.
INSERT INTO print_sink
SELECT order_id, user_id, amount, order_time
FROM fs_csv_source;
```

- **Example 2: Use DataGen to generate simulated data and write the data to HDFS in Parquet columnar format, partitioned by day.**

```
-- Create a DataGen source table.
CREATE TABLE event_source (
  event_id BIGINT,
  event_type STRING,
  event_time TIMESTAMP(3),
  dt STRING
) WITH (
  'connector' = 'datagen',
  'rows-per-second' = '200',
  'fields.event_id.kind' = 'sequence',
  'fields.event_id.start' = '1',
  'fields.event_id.end' = '1000000',
  'fields.event_type.length' = '10',
  'fields.dt.kind' = 'sequence',
  'fields.dt.start' = '20260101',
  'fields.dt.end' = '20260131'
);

-- Create the FileSystem sink result table (Parquet format, partitioned by day).
CREATE TABLE fs_parquet_sink (
  event_id BIGINT,
  event_type STRING,
  event_time TIMESTAMP(3),
  dt STRING
) PARTITIONED BY (dt) WITH (
  'connector' = 'filesystem',
  'path' = 'obs://xxx/data/events/',
  'format' = 'parquet',
  'sink.rolling-policy.file-size' = '256mb',
  'sink.rolling-policy.rollover-interval' = '5min',
  'sink.partition-commit.policy.kind' = 'success-file',
  'sink.partition-commit.trigger' = 'partition-time'
);

-- Submit the job.
INSERT INTO fs_parquet_sink
SELECT event_id, event_type, event_time, dt
FROM event_source;
```

- **Example 3: Stream and monitor new JSON files added to an HDFS directory, and output them to a Print result table in real time.**

```
-- Create a FileSystem source table (JSON format, stream monitoring).
CREATE TABLE fs_json_source (
  log_id BIGINT,
  log_level STRING,
  log_message STRING,
  log_time TIMESTAMP(3)
) WITH (
  'connector' = 'filesystem',
```

 Copyright © Huawei Technologies Co., Ltd.

```
  'path' = 'obs://xxx/data/logs/',
  'format' = 'json',
  'source.monitor-interval' = '10s'
);

-- Create the Print result table.
CREATE TABLE log_print_sink (
  log_id BIGINT,
  log_level STRING,
  log_message STRING,
  log_time TIMESTAMP(3)
) WITH (
  'connector' = 'print',
  'print-identifier' = 'fs_json_logs'
);

-- Submit the job.
INSERT INTO log_print_sink
SELECT log_id, log_level, log_message, log_time
FROM fs_json_source;
```

- **Example 4: Consume order data from Kafka, write the data to HDFS in ORC columnar format with hourly partitioning, and enable automatic file merging.**

```
-- Create a Kafka source table.
CREATE TABLE kafka_order_source (
  order_id STRING,
  user_id BIGINT,
  amount DECIMAL(10, 2),
  status STRING,
  event_time TIMESTAMP(3),
  WATERMARK FOR event_time AS event_time - INTERVAL '5' SECOND,
  dt STRING,
  hh STRING
) WITH (
  'connector' = 'kafka',
  'topic' = 'orders',
  'properties.bootstrap.servers' = 'kafka-broker:9092',
  'properties.group.id' = 'fs_orc_group',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'json',
  'json.ignore-parse-errors' = 'true'
);

-- Create the FileSystem sink result table (ORC format, partitioned by day and hour).
CREATE TABLE fs_orc_sink (
  order_id STRING,
  user_id BIGINT,
  amount DECIMAL(10, 2),
  status STRING,
  event_time TIMESTAMP(3),
  dt STRING,
  hh STRING
) PARTITIONED BY (dt, hh) WITH (
  'connector' = 'filesystem',
  'path' = 'obs://xxx/data/orders_orc/',
  'format' = 'orc',
  'sink.rolling-policy.file-size' = '128mb',
  'sink.rolling-policy.rollover-interval' = '1min',
  'sink.partition-commit.policy.kind' = 'success-file',
  'sink.partition-commit.trigger' = 'partition-time',
  'auto-compaction' = 'true',
  'compaction.file-size' = '256mb'
);

-- Submit the ETL job.
INSERT INTO fs_orc_sink
SELECT order_id, user_id, amount, status, event_time,
       DATE_FORMAT(event_time, 'yyyy-MM-dd') AS dt,
```

 Copyright © Huawei Technologies Co., Ltd.

```
    DATE_FORMAT(event_time, 'HH') AS hh
FROM kafka_order_source;
```

- **Example 5: Read Avro files from HDFS in batch mode, filter the data, and write the filtered results to another HDFS directory in JSON format.**

```
-- Create a FileSystem source table (Avro format).
CREATE TABLE fs_avro_source (
  user_id BIGINT,
  user_name STRING,
  age INT,
  city STRING,
  register_time TIMESTAMP(3)
) WITH (
  'connector' = 'filesystem',
  'path' = 'obs://xxx/data/users_avro/',
  'format' = 'avro'
);

-- Create the FileSystem sink result table (JSON format).
CREATE TABLE fs_json_sink (
  user_id BIGINT,
  user_name STRING,
  age INT,
  city STRING,
  register_time TIMESTAMP(3)
) WITH (
  'connector' = 'filesystem',
  'path' = 'obs://xxx/data/users_json/',
  'format' = 'json',
  'sink.rolling-policy.file-size' = '128mb',
  'sink.rolling-policy.rollover-interval' = '5min'
);

-- Submit the job to filter adult users.
INSERT INTO fs_json_sink
SELECT user_id, user_name, age, city, register_time
FROM fs_avro_source
WHERE age >= 18;
```

- **Example 6: Use DataGen to generate simulated data and write it to the local file system in CSV format. This example is applicable to local test scenarios.**

```
-- Create a DataGen source table.
CREATE TABLE metric_source (
  metric_id BIGINT,
  metric_name STRING,
  metric_value DOUBLE,
  metric_time TIMESTAMP(3)
) WITH (
  'connector' = 'datagen',
  'rows-per-second' = '100',
  'fields.metric_id.kind' = 'sequence',
  'fields.metric_id.start' = '1',
  'fields.metric_id.end' = '100000',
  'fields.metric_name.length' = '15',
  'fields.metric_value.min' = '0.0',
  'fields.metric_value.max' = '100.0'
);

-- Create the FileSystem sink result table (CSV format, local file system).
CREATE TABLE fs_local_csv_sink (
  metric_id BIGINT,
  metric_name STRING,
  metric_value DOUBLE,
  metric_time TIMESTAMP(3)
) WITH (
  'connector' = 'filesystem',
  'path' = 'file:///tmp/flink/metrics/',
  'format' = 'csv',
```

 Copyright © Huawei Technologies Co., Ltd.

```
  'sink.rolling-policy.file-size' = '64mb',
  'sink.rolling-policy.rollover-interval' = '30s'
);

-- Submit the job.
INSERT INTO fs_local_csv_sink
SELECT metric_id, metric_name, metric_value, metric_time
FROM metric_source;
```

1.6 DataGen Connector

Overview

The DataGen connector is a built-in source connector of Flink. It generates random or sequential test data based on rules and does not depend on any external data source.

Table 1-5 Connector details

Category	Details
Supported type	Source table
Running mode	Batch mode and stream mode
Data format	Not applicable
API type	SQL
Whether result table can be updated or deleted	N/A

Application Scenarios

- **Function verification**: Quickly verifies the logic of Flink SQL jobs when no external data source is available.
- **Performance stress testing**: Simulates high-throughput scenarios by controlling the data generation rate and field size.
- **Development and debugging**: Works with the Print or BlackHole connector to build an end-to-end debugging link with no dependency.

Notes and Constraints

- If both **number-of-rows** and the **start** or **end** of the sequence generator are set, the generation stops when either of the conditions is met.

Syntax Structure

```
CREATE TABLE {table_name} (
  {column_name} {column_type},
  ...
  WATERMARK FOR {time_column} AS {watermark_expression}
) WITH (
```

 Copyright © Huawei Technologies Co., Ltd.

```
 'connector' = 'datagen',
 'rows-per-second' = '{Generation rate}',
 'number-of-rows' = '{Total number of rows}',
 'fields.{field_name}.kind' = 'random / sequence',
 'fields.{field_name}.min' = '{Minimum value}',
 'fields.{field_name}.max' = '{Maximum value}',
 'fields.{field_name}.length' = '{String/Set length}',
 'fields.{field_name}.start' = '{Start value}',
 'fields.{field_name}.end' = '{End value}'
 ...
);
```

NOTE

{field_name} in the parameter must be replaced with the actual field name defined in the DDL. For example, if the field name is **user_id**, the parameter is written as **fields.user_id.kind**.

WITH Parameters

Common parameters

Parameter	Mandatory	Type	Default Value	Value Range	Description
connector	Yes	String	None	datagen	Specifies the connector type as DataGen. The value must be fixed at **datagen**. This is the unique identifier for Flink SQL to identify the connector implementation. An incorrect value will cause the table registration to fail.
rows-per-second	No	Long	10000	Positive integer	Controls the number of rows of data generated per second. This parameter takes effect only for the source. During Flink runtime, each parallel subtask independently sends data downstream at this rate. The actual total throughput is calculated as follows: Total throughput = This value × Parallelism.
number-of-rows	No	Long	None (unbounded stream)	Positive integer	Controls the total number of rows of data generated. After this parameter is set, the source automatically stops when the number of rows reaches the specified value, and the stream becomes a bounded stream. This parameter is independent of the **end** value of the sequence generator. The generation stops when either condition is met first. If this parameter is not set, the source continuously generates data and will not automatically stop.

 Copyright © Huawei Technologies Co., Ltd.

Generation policy parameters

Parameter	Mandatory	Type	Default Value	Value Range	Description
fields.{field_name}.kind	No	String	random	random/sequence	Specifies the data generation policy for a specified field. • **random**: A random generator is used to generate random values each time. • **sequence**: A sequence generator is used to increment from **start** to **end**. If not explicitly set, all fields use the random generator by default. Different fields in the same table can have different generation policies specified.

Random generator parameters (kind = random)

Parameter	Mandatory	Type	Default Value	Value Range	Description
fields.{field_name}.min	No	Consistent with the field type	See the default value table of types.	Numeric	Specifies the minimum value generated randomly. This parameter takes effect only for numeric types (**TINYINT**, **SMALLINT**, **INT**, **BIGINT**, **FLOAT**, **DOUBLE**, and **DECIMAL**) and does not take effect for **STRING** and **BOOLEAN** types. It is used as the lower bound when Flink generates random values.
fields.{field_name}.max	No	Consistent with the field type	See the default value table of types.	Numeric	Specifies the maximum value generated randomly. This parameter takes effect only for numeric types. The value of **min** must be less than that of **max**. Otherwise, the generated behavior is undefined. This parameter is used together with **min** to determine the value range of random values.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Type	Default Value	Value Range	Description
fields.{field_name}.length	No	Integer	100	Positive integer	Controls the length (number of characters) of a randomly generated string or the number of elements in a collection. This parameter is used to control the string or byte length for the **STRING**, **VARCHAR**, **CHAR**, **BYTES**, **BINARY**, and **VARBINARY** types, and to control the number of collection elements for the **ARRAY**, **MAP**, and **MULTISET** types. A larger value indicates that more memory is occupied by a single record. If the value is too large, the throughput and GC performance will be affected.
fields.{field_name}.max-past	No	Consistent with the field type	0	Valid duration, for example, 30 seconds or 5 minutes.	Generates a random past time that is no more than *N* away from the current time for the **TIMESTAMP** type. Only the **TIMESTAMP** type is supported.
fields.{field_name}.null-rate	No	Float	None	Greater than or equal to 0 and less than or equal to 1	Controls the probability that this column is generated as null.
fields.{field_name}.var-len	No	BOOLEAN	false	true/false	Controls whether the length of **VARCHAR**, **STRING**, **VARBINARY**, or **BYTES** generated is variable.

 Copyright © Huawei Technologies Co., Ltd.

Default range of each type of random generator:

Data Type	Default Minimum Value	Default Maximum Value	Configurable Parameter
BOOLEAN	-	-	-
TINYINT	-128	127	min, max
SMALLINT	-32768	32767	min, max
INT	-2147483648	2147483647	min, max
BIGINT	-9223372036854775808	9223372036854775807	min, max
FLOAT	Minimum positive value	Maximum positive value	min, max
DOUBLE	Minimum positive value	Maximum positive value	min, max
DECIMAL(p, s)	Determined by the precision	Determined by the precision	min, max
STRING / VARCHAR / CHAR	-	-	length
BYTES / BINARY / VARBINARY	-	-	length
DATE	1970-01-01	Current date	min, max
TIME	00:00:00	23:59:59	min, max
TIMESTAMP	1970-01-01 00:00:00.000	Current timestamp	min, max
TIMESTAMP_LTZ	1970-01-01 00:00:00.000	Current timestamp	min, max
ARRAY<T>	-	-	length (number of elements)
MAP<K, V>	-	-	length (number of key-value pairs)
MULTISET<T>	-	-	length (number of elements)
ROW<...>	-	-	Configured independently for each subfield

 Copyright © Huawei Technologies Co., Ltd.

Sequence generator parameters (kind = sequence)

Parameter	Mandatory	Type	Default Value	Value Range	Description
fields.{field_name}.start	Yes	Consistent with the field type	None	Value range of the numeric type	Specifies the start value of the sequence. This parameter is required only when the sequence generator is used. Flink generates data starting from this value and increments it by 1 each time.
fields.{field_name}.end	Yes	Consistent with the field type	None	Value range of the numeric type	Specifies the end value of the sequence. This parameter is required only when the sequence generator is used. When the sequence reaches this value, the source stops sending data and becomes a bounded stream. In the case of multiple parallelism, the sequence range is evenly allocated to each subtask.

NOTE

- **Data types supported by the sequence generator: TINYINT**, **SMALLINT**, **INT**, **BIGINT**, **FLOAT**, **DOUBLE**, **DECIMAL**, **CHAR**, **VARCHAR**, and **STRING**.
- **CHAR**, **VARCHAR**, and **STRING** are used to convert integers into numeric strings of the **STRING** type.
- The sequence generator does not support data types such as **BOOLEAN**, **DATE**, and **TIMESTAMP**.
- The sequence generator always becomes a bounded stream after reaching the **end** value.

Usage Example

- **Example 1: Quickly create a test table. All fields use the default random generator, and no external dependency is required.**

```
-- Create a DataGen source table.
CREATE TABLE datagen_source (
  id BIGINT,
  data STRING,
  ts TIMESTAMP(3),
  WATERMARK FOR ts AS ts - INTERVAL '5' SECOND
) WITH (
  'connector' = 'datagen'
);
```

 Copyright © Huawei Technologies Co., Ltd.

```
-- Create the Print result table.
CREATE TABLE print_sink (
  id BIGINT,
  data STRING,
  ts TIMESTAMP(3)
) WITH (
  'connector' = 'print'
);

-- Submit the job.
INSERT INTO print_sink
SELECT id, data, ts
FROM datagen_source;
```

- **Example 2: Simulate user behavior data. The ID field is incremented by sequence, and the generation range of other fields is controlled.**

```
-- Create a DataGen source table.
CREATE TABLE user_behavior_source (
  user_id INT,
  user_name STRING,
  score DOUBLE,
  age INT,
  event_time TIMESTAMP(3),
  WATERMARK FOR event_time AS event_time - INTERVAL '5' SECOND
) WITH (
  'connector' = 'datagen',
  'rows-per-second' = '500',
  'fields.user_id.kind' = 'sequence',
  'fields.user_id.start' = '1',
  'fields.user_id.end' = '1000',
  'fields.user_name.length' = '10',
  'fields.score.min' = '0.0',
  'fields.score.max' = '100.0',
  'fields.age.min' = '18',
  'fields.age.max' = '65'
);

-- Create the Print result table.
CREATE TABLE print_sink (
  user_id INT,
  user_name STRING,
  score DOUBLE,
  age INT,
  event_time TIMESTAMP(3)
) WITH (
  'connector' = 'print',
  'print-identifier' = 'user-behavior'
);

-- Submit the job.
INSERT INTO print_sink
SELECT user_id, user_name, score, age, event_time
FROM user_behavior_source;
```

- **Example 3: Generate test data with a fixed number of rows. After the data is generated, the source automatically stops. This example is applicable to batch processing tests.**

```
-- Create a DataGen source table (bounded stream).
CREATE TABLE datagen_bounded_source (
  id INT,
  data STRING
) WITH (
  'connector' = 'datagen',
  'rows-per-second' = '100',
  'number-of-rows' = '500',
  'fields.id.kind' = 'sequence',
  'fields.id.start' = '1',
  'fields.id.end' = '500'
```

 Copyright © Huawei Technologies Co., Ltd.

```
);

-- Create the Print result table.
CREATE TABLE print_sink (
  id INT,
  data STRING
) WITH (
  'connector' = 'print',
  'print-identifier' = 'bounded-output'
);

-- Submit the job.
INSERT INTO print_sink
SELECT id, data
FROM datagen_bounded_source;
```

- **Example 4: Generate test data that contains composite type fields such as ARRAY, MAP, and ROW.**

```
-- Create a DataGen source table (including composite type fields).
CREATE TABLE datagen_complex_source (
  id INT,
  tags ARRAY<STRING>,
  properties MAP<STRING, STRING>,
  address ROW<city STRING, street STRING>,
  payload BYTES
) WITH (
  'connector' = 'datagen',
  'rows-per-second' = '100',
  'fields.id.kind' = 'sequence',
  'fields.id.start' = '1',
  'fields.id.end' = '10000',
  'fields.tags.length' = '3',
  'fields.properties.length' = '5',
  'fields.payload.length' = '50'
);

-- Create the Print result table.
CREATE TABLE print_sink (
  id INT,
  tags ARRAY<STRING>,
  properties MAP<STRING, STRING>,
  address ROW<city STRING, street STRING>,
  payload BYTES
) WITH (
  'connector' = 'print',
  'print-identifier' = 'complex-data'
);

-- Submit the job.
INSERT INTO print_sink
SELECT id, tags, properties, address, payload
FROM datagen_complex_source;
```

- **Example 5: Simulate a high-load scenario and control the data volume and rate to perform performance stress testing.**

```
-- Create a DataGen source table (high-throughput stress test configuration).
CREATE TABLE datagen_pressure_source (
  id BIGINT,
  payload STRING,
  event_time TIMESTAMP(3),
  WATERMARK FOR event_time AS event_time - INTERVAL '1' SECOND
) WITH (
  'connector' = 'datagen',
  'rows-per-second' = '100000',
  'number-of-rows' = '10000000',
  'fields.id.kind' = 'sequence',
  'fields.id.start' = '1',
  'fields.id.end' = '10000000',
  'fields.payload.length' = '1024'
);
```

 Copyright © Huawei Technologies Co., Ltd.

```
-- Create the BlackHole result table.
CREATE TABLE blackhole_sink (
  id BIGINT,
  payload STRING,
  event_time TIMESTAMP(3)
) WITH (
  'connector' = 'blackhole'
);

-- Submit the job to test the pure processing throughput.
INSERT INTO blackhole_sink
SELECT id, payload, event_time
FROM datagen_pressure_source;
```

1.7 BlackHole Connector

Overview

The BlackHole connector is a black hole sink that discards all written data and does not generate any output. It is similar to /dev/null in Unix and is used in scenarios where no actual output is required.

Table 1-6 Connector details

Category	Details
Supported type	Result table
Running mode	Batch mode and stream mode
Data format	Not applicable
API type	SQL
Whether result table can be updated or deleted	Yes

Application Scenarios

- **Performance test**: Tests the throughput of the source and computing layers of Flink SQL jobs, excluding the I/O impact of the sink.
- **Logic verification**: Verifies whether the SQL syntax and job DAG are correctly constructed. You do not need to care about the output result.
- **Development and debugging**: Functions as the final sink placeholder in scenarios where only the source or intermediate processing result needs to be observed.

Notes and Constraints

- It can only be used as a sink and cannot be used as a source.
- All written data will be discarded and cannot be restored.

 Copyright © Huawei Technologies Co., Ltd.

- It is not applicable to data persistence scenarios in the production environment.

Syntax Structure

```
CREATE TABLE {table_name} (
  {column_name} {column_type},
  ...
) WITH (
  'connector' = 'blackhole'
);
```

WITH Parameters

Parameter	Mandatory	Type	Default Value	Value Range	Description
connector	Yes	String	None	blackhole	Specifies that the connector type is BlackHole. This parameter is mandatory and must be set to the fixed value **blackhole**. The Flink runtime loads the corresponding BlackHole SinkFactory based on this value. All written data is directly discarded during runtime, and no I/O operations are generated.

NOTE

For the BlackHole connector, you do not need to configure any other parameters except the connector parameter.

Usage Example

- **Example 1: Create a BlackHole sink table to discard all written data.**

```
-- Create a DataGen source table.
CREATE TABLE datagen_source (
  id BIGINT,
  name STRING,
  `value` DOUBLE
) WITH (
  'connector' = 'datagen',
  'rows-per-second' = '1000',
  'fields.id.kind' = 'sequence',
  'fields.id.start' = '1',
  'fields.id.end' = '100000'
);

-- Create the BlackHole result table.
CREATE TABLE blackhole_sink (
  id BIGINT,
  name STRING,
  `value` DOUBLE
) WITH (
  'connector' = 'blackhole'
);

-- Submit the job.
```

 Copyright © Huawei Technologies Co., Ltd.

```
INSERT INTO blackhole_sink
SELECT id, name, `value`
FROM datagen_source;
```

- **Example 2: Use DataGen to generate high-throughput data and use the BlackHole sink to receive the data, testing the pure processing capacity of the source and computing layers.**

```
-- Create a DataGen source table (high throughput configuration).
CREATE TABLE datagen_source (
  id BIGINT,
  payload STRING,
  event_time TIMESTAMP(3),
  WATERMARK FOR event_time AS event_time - INTERVAL '1' SECOND
) WITH (
  'connector' = 'datagen',
  'rows-per-second' = '100000',
  'fields.id.kind' = 'sequence',
  'fields.id.start' = '1',
  'fields.id.end' = '10000000',
  'fields.payload.length' = '100'
);

-- Create the BlackHole result table.
CREATE TABLE blackhole_sink (
  id BIGINT,
  payload STRING,
  event_time TIMESTAMP(3)
) WITH (
  'connector' = 'blackhole'
);

-- Submit the job to test the pure processing throughput.
INSERT INTO blackhole_sink
SELECT id, payload, event_time
FROM datagen_source;
```

- **Example 3: If the job needs to split and process data, with some data written to Kafka and the other data not requiring output, use BlackHole as a placeholder.**

```
-- Create a DataGen source table.
CREATE TABLE datagen_source (
  user_id INT,
  event_type STRING,
  event_time TIMESTAMP(3)
) WITH (
  'connector' = 'datagen',
  'rows-per-second' = '100',
  'fields.user_id.kind' = 'sequence',
  'fields.user_id.start' = '1',
  'fields.user_id.end' = '100000'
);

-- Main output: Write data to Kafka.
CREATE TABLE kafka_sink (
  user_id INT,
  event_type STRING,
  event_time TIMESTAMP(3)
) WITH (
  'connector' = 'kafka',
  'topic' = 'user-events',
  'properties.bootstrap.servers' = 'kafka-broker:9092',
  'format' = 'json'
);

-- Placeholder output: Discard unnecessary data.
CREATE TABLE blackhole_sink (
  user_id INT,
  event_type STRING,
  event_time TIMESTAMP(3)
```

 Copyright © Huawei Technologies Co., Ltd.

```
) WITH (
  'connector' = 'blackhole'
);

-- Stream distribution: Write login events to Kafka and discard other events.
EXECUTE STATEMENT SET
BEGIN
INSERT INTO kafka_sink
SELECT user_id, event_type, event_time
FROM datagen_source
WHERE event_type = 'login';

INSERT INTO blackhole_sink
SELECT user_id, event_type, event_time
FROM datagen_source
WHERE event_type <> 'login';
END;
```

1.8 Print Connector

Overview

The Print connector outputs data to the standard output (log) of the TaskManager. It is commonly used during development and debugging to quickly view the output results of Flink SQL jobs.

Table 1-7 Connector details

Category	Details
Supported type	Result table
Running mode	Batch mode and stream mode
Data format	Not applicable
API type	SQL
Whether result table can be updated or deleted	Yes

Application Scenarios

- **Development and debugging**: Quickly verifies the output results of Flink SQL processing logic without setting up external storage.
- **Troubleshooting**: Observes the data stream content in real time during job running to locate data processing issues.
- **Data sampling**: Views a small number of data samples through the standard output to check the data format and content.

Notes and Constraints

- It can only be used as a sink and cannot be used as a source.

 Copyright © Huawei Technologies Co., Ltd.

- In scenarios with a large amount of data, the performance may be affected. Therefore, you are not advised to use this connector in the production environment.

Syntax Structure

```
CREATE TABLE {table_name} (
  {column_name} {column_type},
  ...
) WITH (
  'connector' = 'print'
);
```

WITH Parameters

Parameter	Mandatory	Type	Default Value	Value Range	Description
connector	Yes	String	None	print	Specifies that the connector type is Print. This parameter is mandatory and must be set to the fixed value **print**. The Flink runtime loads the corresponding Print SinkFactory based on this value to declare the table as a Print output table.

Usage Example

- **Example 1: Create a minimal Print sink table and directly output data to the TaskManager log.**

```
-- Create a DataGen source table.
CREATE TABLE datagen_source (
  id BIGINT,
  name STRING,
  score DOUBLE
) WITH (
  'connector' = 'datagen',
  'rows-per-second' = '10',
  'fields.id.kind' = 'sequence',
  'fields.id.start' = '1',
  'fields.id.end' = '100'
);

-- Create the Print result table.
CREATE TABLE print_sink (
  id BIGINT,
  name STRING,
  score DOUBLE
) WITH (
  'connector' = 'print'
);

-- Submit the job.
INSERT INTO print_sink
SELECT id, name, score
FROM datagen_source;
```

 Copyright © Huawei Technologies Co., Ltd.

- **Example 2: When there are multiple Print sinks in a job, the output of different data streams is distinguished by the identifier prefix.**

```
-- Create a DataGen source table.
CREATE TABLE datagen_source (
  user_id INT,
  event_type STRING,
  event_time TIMESTAMP(3),
  WATERMARK FOR event_time AS event_time - INTERVAL '5' SECOND
) WITH (
  'connector' = 'datagen',
  'rows-per-second' = '10',
  'fields.user_id.kind' = 'sequence',
  'fields.user_id.start' = '1',
  'fields.user_id.end' = '100'
);

-- Create the Print result table (with an identifier prefix).
CREATE TABLE print_sink_with_id (
  user_id INT,
  event_type STRING,
  event_time TIMESTAMP(3)
) WITH (
  'connector' = 'print',
  'print-identifier' = 'user-event'
);

-- Submit the job.
INSERT INTO print_sink_with_id
SELECT user_id, event_type, event_time
FROM datagen_source;
```

  Example output:

```
user-event> +I[1, login, 2024-01-01 10:00:00.000]
user-event> +I[2, click, 2024-01-01 10:01:00.000]
```

- **Example 3: Use DataGen to generate simulated data and Print to output the results, building a complete zero-dependency test chain.**

```
-- Create a DataGen source table.
CREATE TABLE datagen_source (
  id INT,
  name STRING,
  event_time TIMESTAMP(3),
  WATERMARK FOR event_time AS event_time - INTERVAL '5' SECOND
) WITH (
  'connector' = 'datagen',
  'rows-per-second' = '10',
  'fields.id.kind' = 'sequence',
  'fields.id.start' = '1',
  'fields.id.end' = '100'
);

-- Create the Print result table.
CREATE TABLE print_sink (
  id INT,
  name STRING,
  event_time TIMESTAMP(3)
) WITH (
  'connector' = 'print',
  'print-identifier' = 'debug-output'
);

-- Submit the job: Write source table data to the result table.
INSERT INTO print_sink
SELECT id, name, event_time
FROM datagen_source;
```

 Copyright © Huawei Technologies Co., Ltd.

2 Keywords

For details about Flink V1.20 keywords, see **Reserved Keywords**.

NOTE

If you need to define Flink keywords in DDL statements, add backquotes (`) at both ends of the keywords, for example, **`value`**.

 Copyright © Huawei Technologies Co., Ltd.

3 DDL Statements

3.1 CREATE CATALOG

Overview

The **CREATE CATALOG** statement is used to register an external catalog in the current session. A catalog is the top-level namespace for Flink metadata management, including databases, tables, views, and functions. Currently, AI DataLake supports HiveCatalog, which is used to connect to Hive Metastore to implement data interoperability between Flink and Hive.

Application Scenarios

- **Hive data integration**: HiveCatalog is used to access table data in Hive and perform unified stream and batch processing in Flink SQL.
- **Unified metadata management**: Hive metadata is incorporated into the Flink catalog system to implement unified query and management.
- **Cross-data source query**: After multiple catalogs are registered, you can use the fully qualified name (**catalog.database.table**) to query data across data sources in SQL statements.

Notes and Constraints

- To use HiveCatalog, ensure that the Hive Metastore service is available and the network is connected.
- The **hive-conf-dir** parameter of HiveCatalog must point to a valid Hive configuration directory that contains the **hive-site.xml** file.
- A catalog registered in the current session is valid only in the current session and will not be persisted to the cluster level.
- If a catalog with the same name is registered, the existing catalog will be overwritten. You need to use **IF NOT EXISTS** to avoid conflicts.

Syntax Structure

```
CREATE CATALOG [IF NOT EXISTS] catalog_name
WITH (key1=val1, key2=val2, ...)
```

 Copyright © Huawei Technologies Co., Ltd.

Usage Example

Example 1: Create a HiveCatalog.

```
-- Create a HiveCatalog.
CREATE CATALOG IF NOT EXISTS myhive WITH (
'type' = 'hive',
'default-database' = 'mydb',
'hive-conf-dir' = '/etc/hive/conf'
);
-- Create a DataGen source table.
CREATE TABLE datagen_source (
id BIGINT,
name STRING,
`value` DOUBLE
) WITH (
'connector' = 'datagen',
'rows-per-second' = '100',
'fields.id.kind' = 'sequence',
'fields.id.start' = '1',
'fields.id.end' = '10000',
'fields.name.length' = '8',
'fields.value.min' = '0.0',
'fields.value.max' = '100.0'
);
-- Create the Print result table.
CREATE TABLE print_sink (
id BIGINT,
name STRING,
`value` DOUBLE
) WITH (
'connector' = 'print',
'print-identifier' = 'catalog_output'
);
-- Submit the job.
INSERT INTO print_sink
SELECT id, name, `value`
FROM datagen_source;
```

3.2 CREATE DATABASE

Overview

The **CREATE DATABASE** statement is used to create a database in the current catalog or a specified catalog. A database is a namespace under a catalog, which is used to organize and manage objects such as tables, views, and functions. After a database is created, you can use the **USE** statement to switch to the database for subsequent operations.

Application Scenarios

- **Project isolation**: Create independent databases for different projects or service domains to logically isolate data.
- **Environment management**: Create databases for different environments, such as development, testing, and production environments.
- **Permission partitioning**: Divide access permissions at the database level to control the data visible to different users.

 Copyright © Huawei Technologies Co., Ltd.

Notes and Constraints

- The database name must be unique in the same catalog. If you want to create a database with the same name, use **IF NOT EXISTS**.
- The attributes that can be set in the **WITH** clause depend on the catalog implementation. Different catalogs support different attributes.
- A database can be created only in the current catalog. Cross-catalog database creation is not supported.
- Before deleting a database, ensure that the database contains no tables or other objects, or use **CASCADE** to delete the database in cascading mode.

Syntax Structure

```
CREATE DATABASE [IF NOT EXISTS] [catalog_name.]db_name
[COMMENT database_comment]
WITH (key1=val1, key2=val2, ...)
```

Syntax Description

Parameter	Mandatory	Description
IF NOT EXISTS	No	Specifies that no error is reported when a database with the same name already exists. If this parameter is not specified and a database with the same name already exists, an exception is thrown.
catalog_name	No	Specifies the name of the target catalog. If this parameter is not specified, the current catalog is used.
db_name	Yes	Database name, which must be unique in the catalog. You are advised to use lowercase letters and underscores (_) in the name.
COMMENT	No	Comment of the database, which helps understand the database usage and service.
WITH	No	Sets database-level attributes in key-value pairs. The available attributes depend on the catalog implementation.

Usage Example

Example 1: Create a database.

```
CREATE DATABASE IF NOT EXISTS mydb COMMENT 'AI DataLake service database';
```

Example 2: Create a Paimon database with the WITH attribute.

```
CREATE DATABASE IF NOT EXISTS paimon_db
COMMENT 'Paimon metadata database'
WITH (
 'location' = 'obs://bucket/path/paimon-warehouse/paimon_db'
);
```

 Copyright © Huawei Technologies Co., Ltd.

3.3 CREATE TABLE

Overview

The **CREATE TABLE** statement is used to register a table in the current or specified catalog, and defines the table structure (physical columns, computed columns, and metadata columns), time attributes (watermarks), constraints (primary keys), and connector configurations (**WITH** clause). The registered table can be used as a source table or result table in SQL queries. AI DataLake supports the standard Flink SQL syntax for creating tables, including core capabilities such as partitioned tables and table creation using **LIKE**.

Application Scenarios

- **Data access**: Uses **CREATE TABLE** to define a mapping table for external data sources (such as Kafka and HDFS) to read and write data.
- **Streaming ETL**: Defines a source table with watermarks and computed columns, and uses window functions to perform real-time data transformations.
- **Partition management**: Uses **PARTITIONED BY** to create partitioned tables, optimizing query performance in scenarios with large data volumes.
- **Table structure reuse**: Quickly creates a new table based on an existing table using the **LIKE** clause, reducing repeated definitions.

Notes and Constraints

- AI DataLake does not support the **CREATE TEMPORARY TABLE** syntax. All tables are persistent tables.
- The primary key constraint is declared as **NOT ENFORCED**. Flink does not verify data uniqueness; this must be ensured by the external storage.
- Only one watermark can be defined for a table, meaning only one event time field can be declared.
- **PARTITIONED BY** is supported only when the connector supports partitioning (for example, Hive and Paimon). Connectors such as Kafka do not support partitioned tables.
- A computed column cannot reference other computed columns. It can only reference physical columns and metadata columns.
- The availability of metadata columns depends on the connector implementation. Different connectors support different metadata.
- The **INCLUDING**/**EXCLUDING** option of the **LIKE** clause must match the structure of the source table. An incorrect combination will cause table creation to fail.

Syntax Structure

Complete syntax

```
CREATE TABLE [IF NOT EXISTS] [catalog_name.][db_name.]table_name (
  { <physical_column_definition> | <metadata_column_definition> | <computed_column_definition> }
  [ , ...n]
```

 Copyright © Huawei Technologies Co., Ltd.

```
  [ <watermark_definition> ]
  [ <table_constraint> ]
  [ , ...n]
)
[COMMENT table_comment]
[PARTITIONED BY (partition_column_name1, partition_column_name2, ...)]
WITH (key1=val1, key2=val2, ...)
[ LIKE source_table [( <like_options> )] ]
```

Column definition syntax

```
-- Physical column
<physical_column_definition>:
  column_name column_type [ <column_constraint> ] [COMMENT column_comment]

<column_constraint>:
  [CONSTRAINT constraint_name] PRIMARY KEY NOT ENFORCED

-- Metadata column
<metadata_column_definition>:
  column_name column_type METADATA [ FROM metadata_key ] [ VIRTUAL ]

-- Computed column
<computed_column_definition>:
  column_name AS computed_column_expression [COMMENT column_comment]
```

Watermark definition syntax

```
<watermark_definition>:
  WATERMARK FOR rowtime_column_name AS watermark_strategy_expression
```

Primary key constraint syntax

```
<table_constraint>:
  [CONSTRAINT constraint_name] PRIMARY KEY (column_name, ...) NOT ENFORCED
```

LIKE option syntax

```
<like_options>:
  {
    { INCLUDING | EXCLUDING } { ALL | CONSTRAINTS | PARTITIONS }
  | { INCLUDING | EXCLUDING | OVERWRITING } { GENERATED | OPTIONS | WATERMARKS }
  }[, ...]
```

Parameter Description

Main clause parameters

Parameter	Mandatory	Description
IF NOT EXISTS	No	Specifies that no error is reported and the creation is skipped if a table with the same name already exists. If this parameter is not specified and a table with the same name exists, an exception is thrown.
catalog_name.db_name	No	Specifies the catalog and database name of the table. If not specified, the current catalog and database are used.
table_name	Yes	Specifies the name of the table to be created. It must be unique in the same database.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Description
COMMENT table_comment	No	Adds comments to a table.
PARTITIONED BY	No	Specifies the partition column. The table is stored in different directories based on the partition column values. This parameter takes effect only when the connector supports partitioning. The partition column must be a physical column defined in the table.
WITH (key1=val1, ...)	Yes	Specifies the connector configuration items of the table, which are provided in the form of key-value pairs. The **connector** parameter is mandatory and determines the data source or destination.
LIKE source_table	No	Creates a table based on the structure of an existing table. The **INCLUDING**, **EXCLUDING**, and **OVERWRITING** parameters can be used to control the structure inheritance policy. The table structure can be reused to avoid repeated definition.

Column type parameters

Column Type	Keyword	Description
Physical column	No special keyword	Defines the columns that store data in the physical medium. This is the most commonly used column definition method. The order of physical columns corresponds to the order of fields in the data source.
Computed column	AS	A virtual column generated by an expression and is not physically stored in the table. It is often used to define the processing time (**AS PROCTIME()**) or derived fields. Only physical columns and metadata columns can be referenced, and other computed columns cannot be referenced.
Metadata column	METADATA	Accesses metadata information of the connector data source, such as the timestamp and offset of Kafka. The **VIRTUAL** flag indicates that the metadata is read-only and is not persisted. The availability depends on the connector implementation.

 Copyright © Huawei Technologies Co., Ltd.

Watermark parameters

Parameter	Mandatory	Description
rowtime_column_name	Yes	Specifies the event time column, which must be of the **TIMESTAMP(3)** or **TIMESTAMP_LTZ(3)** type. Only one watermark can be defined for a table.
watermark_strategy_expression	Yes	Watermark generation strategy expression, which is used to calculate the timestamp of the watermark. A common strategy is **rowtime_column - INTERVAL 'n' SECOND**, indicating that a maximum of *n* seconds of out-of-order is allowed.

Primary key constraint parameters

Parameter	Mandatory	Description
constraint_name	No	Constraint name, which is optional.
column_name, ...	Yes	One or more columns that are included in the primary key. These columns cannot be null.
NOT ENFORCED	Yes	Flink does not forcibly verify the uniqueness of primary keys. The uniqueness of primary keys must be ensured by external storage. This keyword is mandatory and cannot be omitted. The primary key is used to determine the semantics of the upsert connector.

LIKE option parameters

Option	Optional Policy	Description
ALL	INCLUDING / EXCLUDING	Controls whether to include all features of the source table. **INCLUDING ALL** indicates that all features are inherited, and **EXCLUDING ALL** indicates that all features are excluded.

 Copyright © Huawei Technologies Co., Ltd.

Option	Optional Policy	Description
CONSTRAINTS	INCLUDING / EXCLUDING	Controls whether to include constraints such as primary keys. **INCLUDING CONSTRAINTS** indicates that constraints are inherited, and **EXCLUDING CONSTRAINTS** indicates that constraints are not inherited.
PARTITIONS	INCLUDING / EXCLUDING	Controls whether to include partition information. **INCLUDING PARTITIONS** indicates that partition definitions are inherited, and **EXCLUDING PARTITIONS** indicates that partition definitions are not inherited.
GENERATED	INCLUDING / EXCLUDING / OVERWRITING	Controls the inheritance of computed columns and metadata columns. **INCLUDING** indicates that the columns are inherited, **EXCLUDING** indicates that the columns are excluded, and **OVERWRITING** indicates that the computed columns in the source table are overwritten by the newly defined columns.
OPTIONS	INCLUDING / EXCLUDING / OVERWRITING	Controls the inheritance of the **WITH** options. **INCLUDING OPTIONS** indicates that all connector configurations are inherited, **EXCLUDING OPTIONS** indicates that all connector configurations are not inherited, and **OVERWRITING OPTIONS** indicates that the new configuration overwrites the items with the same name.
WATERMARKS	INCLUDING / EXCLUDING / OVERWRITING	Controls the inheritance of watermark definitions. **INCLUDING WATERMARKS** indicates that watermarks are inherited, **EXCLUDING WATERMARKS** indicates that watermarks are not inherited, and **OVERWRITING WATERMARKS** indicates that the new definition overwrites the existing definition.

Usage Example

AI DataLake supports multiple connectors, each of which has specific usage examples. For details, see **1.1 Supported Connectors** and view the usage examples in the documentation of each connector.

 Copyright © Huawei Technologies Co., Ltd.

3.4 CREATE VIEW

Overview

CREATE VIEW is used to create a virtual table based on a **SELECT** query. A view does not store data. Instead, it dynamically executes the underlying query each time it is accessed. Views can be used to simplify complex queries, provide a logical data abstraction layer, and implement access control.

Application Scenarios

- **Simplified query**: Complex queries such as multi-table JOIN and aggregation are encapsulated into views, and the view names are directly referenced in subsequent SQL statements.
- **Data masking**: Creates a view that contains only public fields and hides sensitive columns.
- **Logical layering**: In the ETL process, defines intermediate processing logic using views to improve SQL readability and maintainability.

Notes and Constraints

- AI DataLake does not support the **CREATE TEMPORARY VIEW** syntax.
- A view is read-only. Data cannot be inserted into or updated in a view.
- The table or view referenced in the view definition must exist. Otherwise, the view fails to be created.
- Views do not support table attribute clauses such as **PARTITIONED BY** and **WITH**.
- When the underlying table structure changes (for example, a column is deleted), the view may become invalid.

Syntax Structure

```
CREATE VIEW [IF NOT EXISTS] [catalog_name.][db_name.]view_name
[( columnName [, columnName ]* )]
[COMMENT view_comment]
AS query_expression
```

Syntax Description

Parameter	Mandatory	Description
IF NOT EXISTS	No	Specifies that no error is reported when a view with the same name already exists. If this parameter is not specified, an exception is thrown if a view with the same name already exists.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Description
view_name	Yes	Specifies the view name, which must be unique in the database. You can specify the **catalog_name** and **db_name** prefixes.
columnName	No	Specifies the list of column names in the customized view. If this parameter is omitted, the column names in the query result are used. The number of column names must be the same as that of columns in the query result.
COMMENT	No	Specifies the comment of the view, which helps understand the purpose and service meaning of the view.
query_expression	Yes	Specifies the **SELECT** query statement that defines the view content. The statement can reference tables, other views, and functions.

Usage Example

Example 1: Create a view.

```
-- Create a DataGen source table.
CREATE TABLE datagen_source (
user_id BIGINT,
product_id BIGINT,
amount DECIMAL(10, 2),
category STRING,
ts TIMESTAMP(3),
WATERMARK FOR ts AS ts - INTERVAL '5' SECOND
) WITH (
'connector' = 'datagen',
'rows-per-second' = '100',
'fields.user_id.kind' = 'sequence',
'fields.user_id.start' = '1',
'fields.user_id.end' = '10000',
'fields.product_id.kind' = 'sequence',
'fields.product_id.start' = '100',
'fields.product_id.end' = '200',
'fields.category.length' = '6'
);
-- Create the Print result table.
CREATE TABLE print_sink (
user_id BIGINT,
total_amount DECIMAL(10, 2)
) WITH (
'connector' = 'print',
'print-identifier' = 'view_output'
);
-- Create a view to aggregate amounts by user.
CREATE VIEW IF NOT EXISTS user_amount_view AS
SELECT user_id, SUM(amount) AS total_amount
FROM datagen_source
GROUP BY user_id;
-- Submit the job.
INSERT INTO print_sink
```

 Copyright © Huawei Technologies Co., Ltd.

```
SELECT user_id, total_amount
FROM user_amount_view;
```

3.5 CREATE FUNCTION

Overview

The **CREATE FUNCTION** statement is used to register user-defined functions (UDFs) in the current catalog, including scalar functions, table functions, and aggregate functions. The registered functions can be used in SQL queries like built-in functions.

Application Scenarios

- **Business logic encapsulation**: Complex business computing logic is encapsulated into UDFs and invoked concisely in SQL.
- **Format conversion**: User-defined data format parsing functions are used to process non-standard data such as JSON and XML.
- **Encryption and anonymization**: User-defined encryption and anonymization algorithms are implemented to meet data security requirements.

Notes and Constraints

- AI DataLake does not support the **TEMPORARY** keyword. Therefore, temporary functions cannot be created.
- The JAR file of the user-defined function must be uploaded to the cluster in advance to ensure that the class path is accessible.
- The function class must implement the UDF interface corresponding to Flink (such as **ScalarFunction**, **TableFunction**, and **AggregateFunction**).
- The function name must be unique in the same database.

Syntax Structure

```
CREATE [SYSTEM] FUNCTION
[IF NOT EXISTS] [catalog_name.][db_name.]function_name
AS class_identifier
[LANGUAGE JAVA|SCALA|PYTHON]
```

Syntax Description

Parameter	Mandatory	Description
SYSTEM	No	Specifies that the function is a system function. The system function has a higher resolution priority than the built-in function and can override the behavior of the built-in function with the same name.
IF NOT EXISTS	No	Specifies that no error is reported when a function with the same name already exists. If this parameter is not specified, an exception is thrown when a function with the same name already exists.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Description
function_name	Yes	Specifies the function name, which must be unique in the database. You can specify the **catalog_name** and **db_name** prefixes.
class_identifier	Yes	Specifies the fully qualified class name of the UDF implementation class. The class must implement the UDF interface corresponding to Flink, and the JAR file must be available in the cluster.
LANGUAGE	No	Specifies the implementation language of the UDF. The supported languages are Java, Scala, and Python. The default language is Java. For Python UDFs, you need to configure the Python environment.

Usage Example

Example: Create a Java UDF.

```
-- Create a DataGen source table.
CREATE TABLE datagen_source (
id BIGINT,
raw_text STRING
) WITH (
'connector' = 'datagen',
'rows-per-second' = '10',
'fields.id.kind' = 'sequence',
'fields.id.start' = '1',
'fields.id.end' = '1000',
'fields.raw_text.length' = '10'
);
-- Create the Print result table.
CREATE TABLE print_sink (
id BIGINT,
processed_text STRING
) WITH (
'connector' = 'print'
);
-- Register the UDF.
CREATE FUNCTION IF NOT EXISTS my_udf AS 'com.example.MyScalarFunction';
-- Submit the job.
INSERT INTO print_sink
SELECT id, my_udf(raw_text) AS processed_text
FROM datagen_source;
```

3.6 ALTER Statement

Overview

The **ALTER** statement is used to modify the definition of an object that has been registered with the catalog, including tables, views, databases, functions, and catalogs. You can use the **ALTER** statement to perform schema evolution, modify attributes, and rename objects without rebuilding them.

 Copyright © Huawei Technologies Co., Ltd.

Application Scenarios

- **Schema evolution**: When service requirements change, you can use **ALTER TABLE ADD**/**DROP COLUMN** to adjust the table structure online without rebuilding the table.
- **Parameter tuning**: To adjust connector configuration parameters during job runtime, use **ALTER TABLE SET** to modify the **WITH** attributes.
- **Object management**: Renames tables or views to correct non-standard naming issues.

Notes and Constraints

- By default, columns added using **ALTER TABLE** are appended to the end of the table. Specifying the position of a column is not supported.
- When modifying a watermark or primary key constraint, ensure that the modification is compatible with the existing column definitions.
- The attributes that can be modified using **ALTER DATABASE** depend on the catalog implementation. Different catalogs support different attributes.
- **ALTER VIEW** only supports renaming the view and modifying its attributes. It does not support modifying the query definition of the view.
- Executing **ALTER** on a table that is currently being used by a job may cause the job to behave abnormally. It is recommended that the **ALTER** operation be performed after the job has been stopped.
- **ALTER TABLE** only modifies metadata. Whether the physical table structure can be modified depends on the connector implementation.

Syntax Structure

ALTER TABLE

-- Rename a table.

```
ALTER TABLE [catalog_name.][db_name.]table_name RENAME TO new_table_name;
```

-- Modify table attributes.

```
ALTER TABLE [catalog_name.][db_name.]table_name SET (key1=val1, key2=val2, ...);
```

-- Add a column.

```
ALTER TABLE [catalog_name.][db_name.]table_name ADD (col_name col_type [COMMENT
col_comment], ...);
```

-- Drop a column.

```
ALTER TABLE [catalog_name.][db_name.]table_name DROP (col_name, ...);
```

-- Add a watermark.

```
ALTER TABLE [catalog_name.][db_name.]table_name ADD WATERMARK FOR rowtime_col AS
watermark_expr;
```

-- Drop a watermark.

```
ALTER TABLE [catalog_name.][db_name.]table_name DROP WATERMARK;
```

-- Add a primary key.

```
ALTER TABLE [catalog_name.][db_name.]table_name ADD PRIMARY KEY (col_name, ...) NOT ENFORCED;
```

 Copyright © Huawei Technologies Co., Ltd.

-- Drop a primary key.

```
ALTER TABLE [catalog_name.][db_name.]table_name DROP PRIMARY KEY;
```

-- Rename a column.

```
ALTER TABLE [catalog_name.][db_name.]table_name RENAME col_name TO new_col_name;
```

-- Modify a column type.

```
ALTER TABLE [catalog_name.][db_name.]table_name MODIFY (col_name new_col_type [COMMENT
col_comment], ...);
```

ALTER VIEW

-- Rename a view.

```
ALTER VIEW [catalog_name.][db_name.]view_name RENAME TO new_view_name;
```

-- Alter view attributes.

```
ALTER VIEW [catalog_name.][db_name.]view_name SET (key1=val1, key2=val2, ...);
```

ALTER DATABASE

-- Alter database attributes.

```
ALTER DATABASE [catalog_name.]db_name SET (key1=val1, key2=val2, ...);
```

ALTER FUNCTION

-- Alter the function definition.

```
ALTER [SYSTEM] FUNCTION [IF EXISTS] [catalog_name.][db_name.]function_name
AS class_identifier
[LANGUAGE JAVA|SCALA|PYTHON];
```

ALTER CATALOG

-- Alter catalog attributes.

```
ALTER CATALOG catalog_name SET (key1=val1, key2=val2, ...);
```

Usage Example

Example: Add columns and alter attributes using ALTER TABLE.

```
-- ALTER TABLE paimon_append_source ADD (email STRING COMMENT 'User email address');
-- ALTER TABLE paimon_append_source SET ('rows-per-second' = '200');
```

3.7 DROP Statements

Overview

DROP statements are used to delete registered objects from a catalog, including tables, databases, views, functions, and catalogs. The deletion operation removes the metadata definition of the object. For some connectors (such as Hive), the underlying physical data is also deleted.

Application Scenarios

- **Resource cleanup**: Deletes tables, views, or databases that are no longer used to release metadata and storage resources.

 Copyright © Huawei Technologies Co., Ltd.

- **Environment reset**: Deletes temporary objects during development or testing to restore the initial status.
- **Function management**: Deregisters custom functions that are no longer used to avoid naming conflicts.

Notes and Constraints

- **DROP TABLE** deletes only metadata. Whether to delete physical data depends on the connector implementation. HiveCatalog deletes both physical data and metadata by default, while other connectors may only delete metadata.
- By default, **DROP DATABASE** uses the **RESTRICT** mode. If the database is not empty, an exception is thrown. You can use **CASCADE** to delete all objects in the database in cascading mode.
- Deleting a table or view that is being used by a job may cause job exceptions. You are advised to perform this operation after the job is stopped.
- **DROP CATALOG** only removes the catalog registration information in the session and does not affect the data in the external metastore.
- After a function is deleted using **DROP FUNCTION**, the SQL query that references the function cannot be executed.

Syntax Structure

DROP TABLE

```
DROP TABLE [IF EXISTS] [catalog_name.][db_name.]table_name;
```

DROP DATABASE

```
DROP DATABASE [IF EXISTS] [catalog_name.]db_name [ (RESTRICT | CASCADE) ];
```

DROP VIEW

```
DROP VIEW [IF EXISTS] [catalog_name.][db_name.]view_name;
```

DROP FUNCTION

```
DROP [SYSTEM] FUNCTION [IF EXISTS] [catalog_name.][db_name.]function_name;
```

DROP CATALOG

```
DROP CATALOG [IF EXISTS] catalog_name;
```

Syntax Description

Parameter	Mandatory	Description
IF EXISTS	No	Specifies that no error is reported when the object does not exist. If this parameter is not specified and the object does not exist, an exception is thrown. You are advised to always use this parameter to avoid script interruption.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory	Description
RESTRICT	No	Specifies the default mode of **DROP DATABASE**. If the database is not empty, an exception is thrown to prevent the database containing objects from being deleted by mistake.
CASCADE	No	Cascades the deletion of all tables, views, and functions in the database. Exercise caution when using this parameter and ensure that the deletion scope is correct.
SYSTEM	No	Deletes a system function. The system function is parsed with a higher priority than the built-in function. After the system function is deleted, the default behavior of the built-in function is restored.

Usage Example

Example: DROP TABLE and DROP DATABASE

```
-- DROP TABLE IF EXISTS my_table;
-- DROP DATABASE IF EXISTS my_db CASCADE;
-- DROP VIEW IF EXISTS my_view;
-- DROP FUNCTION IF EXISTS my_udf;
```

 Copyright © Huawei Technologies Co., Ltd.

4 DML Statements

4.1 INSERT INTO

Overview

The **INSERT INTO** statement is a core data writing statement in Flink SQL. It is used to write the result of a **SELECT** query or explicitly specified values to a target table (sink table). AI DataLake supports **INSERT INTO**, which appends data to the target table without overwriting existing data. This is the most commonly used write mode and applies to both streaming and batch processing scenarios.

Additionally, AI DataLake supports using the **STATEMENT SET** syntax to combine multiple **INSERT** statements into a single Flink job for execution. This allows multiple write operations to share the same directed acyclic graph (DAG), thereby reducing TaskManager resource overhead. This mode is suitable for multi-sink data distribution scenarios.

Application Scenarios

- **Real-time data ingestion**: Real-time data from message queues such as Kafka can be written to storage systems like Paimon or Hive using **INSERT INTO** to build a real-time data warehouse.
- **Multi-sink data distribution**: After ETL processing, data from the same source needs to be written to multiple target tables (for example, detail tables and aggregation tables simultaneously). Using **STATEMENT SET** combines these operations into one job, reducing resource consumption.
- **Dynamic parameter tuning**: You can use table hints to dynamically adjust connector parameters in **INSERT** statements, allowing the table to adapt to different running environments without modifying the table DDL.

Notes and Constraints

- All **INSERT** statements in a **STATEMENT SET** must share the same source table. Different **INSERT** statements cannot use completely independent source tables. Otherwise, they may not be combined into the same DAG.
- Multiple **INSERT** statements in a **STATEMENT SET** are submitted as a whole. You cannot individually control the start or stop of a single **INSERT** statement.

 Copyright © Huawei Technologies Co., Ltd.

- The number and types of columns in the **SELECT** clause of the **INSERT** statement must match the column definitions of the target table, or the corresponding columns must be specified in the explicit **column_list**.
- When writing to a partition, the dynamic partition columns must appear at the end of the **SELECT** clause and be in the same order as declared in the **PARTITION** clause.
- Table hints (**/*+ OPTIONS(...) */**) take effect only for the table instance in the current **INSERT** statement and do not modify the persistent definition of the table in the catalog.

Syntax Structure

INSERT INTO single-sink write

```
INSERT INTO [catalog_name.][db_name.]table_name
[PARTITION part_spec]
[column_list]
select_statement | VALUES (value [, value ...]) [, (value [, value ...]) ...];
part_spec ::= (key1=value1, key2=value2, ...)
column_list ::= (col1 [, col2, ...])
```

INSERT INTO multi-sink write

```
EXECUTE STATEMENT SET
BEGIN
INSERT INTO sink_table1 select_statement1;
INSERT INTO sink_table2 select_statement2;
[...]
END;
```

NOTE

All **INSERT** statements between **EXECUTE STATEMENT SET** and **END** are combined into a single Flink job. Multiple **INSERT** statements share the source operator and are executed in the same DAG, effectively reducing TaskManager resource usage.

Dynamic parameter passing using table hints

```
INSERT INTO table_name /*+ OPTIONS('key1'='value1', 'key2'='value2') */
select_statement;
```

Usage Example

Example 1: INSERT INTO single-sink write

Use the DataGen connector to generate simulated sensor data and use the **INSERT INTO** statement to write the data to the Print connector for output.

```
-- Create a sensor data source table.
CREATE TABLE sensor_source (
sensor_id INT,
temperature DOUBLE,
humidity DOUBLE,
record_time TIMESTAMP(3),
WATERMARK FOR record_time AS record_time - INTERVAL '5' SECOND
) WITH (
'connector' = 'datagen',
'rows-per-second' = '10',
'fields.sensor_id.kind' = 'random',
'fields.sensor_id.min' = '1',
'fields.sensor_id.max' = '100',
'fields.temperature.min' = '15.0',
'fields.temperature.max' = '40.0',
```

 Copyright © Huawei Technologies Co., Ltd.

```
'fields.humidity.min' = '30.0',
'fields.humidity.max' = '90.0'
);
-- Create the sensor data result table.
CREATE TABLE sensor_sink (
sensor_id INT,
temperature DOUBLE,
humidity DOUBLE
) WITH (
'connector' = 'print',
'print-identifier' = 'sensor_data'
);
-- Submit the job: Filter sensor data with temperature exceeding 30°C.
INSERT INTO sensor_sink
SELECT sensor_id, temperature, humidity
FROM sensor_source
WHERE temperature > 30;
```

Example 2: INSERT INTO multi-sink write (STATEMENT SET)

Use the DataGen connector to generate order data, and use **STATEMENT SET** to write the data to the detail table and the aggregate statistics table respectively. The two **INSERT** statements are executed in the same Flink job.

-- Create an order data source table.

```
CREATE TABLE order_source (
order_id BIGINT,
customer_id BIGINT,
product_id BIGINT,
amount DECIMAL(10, 2),
order_time TIMESTAMP(3),
WATERMARK FOR order_time AS order_time - INTERVAL '5' SECOND
) WITH (
'connector' = 'datagen',
'rows-per-second' = '100',
'fields.order_id.kind' = 'sequence',
'fields.order_id.start' = '1',
'fields.order_id.end' = '1000000',
'fields.customer_id.kind' = 'random',
'fields.customer_id.min' = '1',
'fields.customer_id.max' = '5000',
'fields.product_id.kind' = 'random',
'fields.product_id.min' = '1',
'fields.product_id.max' = '200',
'fields.amount.min' = '10.00',
'fields.amount.max' = '1000.00'
);
-- Create the order details result table.
CREATE TABLE order_detail_sink (
order_id BIGINT,
customer_id BIGINT,
amount DECIMAL(10, 2)
) WITH (
'connector' = 'print',
'print-identifier' = 'order_detail'
);
-- Create the customer aggregation result table.
CREATE TABLE customer_agg_sink (
customer_id BIGINT,
order_count BIGINT,
total_amount DECIMAL(10, 2)
) WITH (
'connector' = 'print',
'print-identifier' = 'customer_agg'
);
-- Submit the job: multi-sink writes, two INSERT statements share the same DAG.
EXECUTE STATEMENT SET
BEGIN
```

 Copyright © Huawei Technologies Co., Ltd.

```
INSERT INTO order_detail_sink
  SELECT order_id, customer_id, amount
FROM order_source;
INSERT INTO customer_agg_sink
  SELECT customer_id, COUNT(*) AS order_count, SUM(amount) AS total_amount
FROM order_source
GROUP BY customer_id;
END;
```

 Copyright © Huawei Technologies Co., Ltd.

5 Queries Statements

Queries Statement	Related Documentation
Hints	**SQL Hints**
WITH clause	**WITH clause**
SELECT and WHERE clauses	**SELECT & WHERE clause**
SELECT DISTINCT	**SELECT DISTINCT**
Window function	**Windowing table-valued functions (Windowing TVFs)**
Window aggregation	**Window Aggregation**
Grouped aggregation	**Group Aggregation**
Over aggregation	**Over Aggregation**
Join	**Joins**
Window join	**Window Join**
Set operation	**Set Operations**
ORDER BY clause	**ORDER BY clause**
LIMIT clause	**LIMIT clause**
Top-N	**Top-N**
Window top-N	**Window Top-N**

 Copyright © Huawei Technologies Co., Ltd.

Queries Statement	Related Documentation
Deduplication	**Deduplication**
Window deduplication	**Window Deduplication**
Pattern recognition	**Pattern Recognition**
Time travel	**Time Travel** **NOTE** Currently, only Paimon catalogs support time travel. For details, see **Time Travel**.

 Copyright © Huawei Technologies Co., Ltd.

6 Built-in Functions

6.1 Comparison Functions

Overview

Comparison functions are used to compare two expressions and return a result of the BOOLEAN type. AI DataLake supports a complete set of comparison operations defined in the SQL standard, including equality comparison, magnitude comparison, range comparison, pattern matching, and null value judgment.

For more information, see **Comparison Functions**.

Application Scenarios

- **WHERE filtering**: Uses comparison conditions to filter data rows in a query.
- **JOIN**: Uses equality comparison in JOIN conditions to associate different tables.
- **Conditional branching**: Uses the comparison result in **CASE WHEN** to control business logic.
- **Data verification**: Uses comparison operations to check the validity and consistency of data.

Notes and Constraints

- The data types of the two ends of a comparison operation must be compatible. Otherwise, an explicit CAST conversion is required.
- The result of comparing **NULL** with any value is **NULL** (not **TRUE** or **FALSE**). Use **IS NULL** to determine the result.
- **BETWEEN** is a closed interval, including the boundary values at both ends. That is, **value1 BETWEEN value2 AND value3** is equivalent to **value1 >= value2 AND value1 <= value3**.
- In **LIKE**, the percent sign (%) matches any sequence of characters (including an empty sequence), and the underscore (_) matches exactly one character. To match a wildcard character itself, use the **ESCAPE** clause for escaping.

 Copyright © Huawei Technologies Co., Ltd.

If the IN list contains **NULL** and **value1** is not in the non-**NULL** values of the list, the result is **NULL** instead of **FALSE**.

Syntax Structure

Syntax	Return Type	Description	Example
value1 = value2	BOOLEAN	Equality comparison. If **value1** is equal to **value2**, **TRUE** is returned. Otherwise, **FALSE** is returned. The types of **value1** and **value2** must be compatible. If either operand is **NULL**, **NULL** is returned.	`SELECT 1 = 1; -- TRUE`
value1 <> value2	BOOLEAN	Inequality comparison. If **value1** is not equal to **value2**, **TRUE** is returned. Otherwise, **FALSE** is returned. The types of **value1** and **value2** must be compatible. If either operand is **NULL**, **NULL** is returned.	`SELECT 1 <> 2; -- TRUE`
value1 != value2	BOOLEAN	Inequality comparison. It is semantically equivalent to **<>**. If **value1** is not equal to **value2**, **TRUE** is returned. If either operand is **NULL**, **NULL** is returned.	`SELECT 1 != 2; -- TRUE`
value1 > value2	BOOLEAN	Greater than comparison. If **value1** is greater than **value2**, **TRUE** is returned. Otherwise, **FALSE** is returned. The types of **value1** and **value2** must be compatible and sortable. If either operand is **NULL**, **NULL** is returned.	`SELECT 3 > 2; -- TRUE`

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
value1 >= value2	BOOLEAN	Greater than or equal to comparison. If **value1** is greater than or equal to **value2**, **TRUE** is returned. Otherwise, **FALSE** is returned. The types of **value1** and **value2** must be compatible and sortable. If either operand is **NULL**, **NULL** is returned.	`SELECT 2 >= 2; -- TRUE`
value1 < value2	BOOLEAN	Less than comparison. If **value1** is less than **value2**, **TRUE** is returned. Otherwise, **FALSE** is returned. The types of **value1** and **value2** must be compatible and sortable. If either operand is **NULL**, **NULL** is returned.	`SELECT 1 < 2; -- TRUE`
value1 <= value2	BOOLEAN	Less than or equal to comparison. If **value1** is less than or equal to **value2**, **TRUE** is returned. Otherwise, **FALSE** is returned. The types of **value1** and **value2** must be compatible and sortable. If either operand is **NULL**, **NULL** is returned.	`SELECT 2 <= 3; -- TRUE`
value IS NULL	BOOLEAN	Determines whether a value is **NULL**. If **value** is NULL, **TRUE** is returned. Otherwise, **FALSE** is returned. This function is not affected by the rule that **NULL** is returned when **NULL** is involved in a common comparison. It is the standard method for determining whether a value is **NULL**.	`SELECT NULL IS NULL; -- TRUE`

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
value IS NOT NULL	BOOLEAN	Determines whether a value is not **NULL**. If **value** is not **NULL**, **TRUE** is returned. Otherwise, **FALSE** is returned. This is the logical negation of **IS NULL**.	SELECT 1 IS NOT NULL; -- TRUE
value1 BETWEEN value2 AND value3	BOOLEAN	Range judgment (closed interval). If **value1** is within the closed interval [**value2**, **value3**], **TRUE** is returned, which is equivalent to **value1 >= value2 AND value1 <= value3**, including both boundary values. **value2** must be less than or equal to **value3**. Otherwise, the result is always **FALSE**. If any operand is **NULL**, **NULL** is returned.	SELECT 2 BETWEEN 1 AND 3; -- TRUE
value1 NOT BETWEEN value2 AND value3	BOOLEAN	Out-of-range judgment. If **value1** is not within the closed interval [value2, value3], **TRUE** is returned, which is equivalent to **value1 < value2 OR value1 > value3**. This is the logical negation of **BETWEEN**. If any operand is **NULL**, **NULL** is returned.	SELECT 5 NOT BETWEEN 1 AND 3; -- TRUE

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
value1 LIKE value2	BOOLEAN	Pattern matching. Returns **TRUE** if **value1** matches pattern **value2**. **value1** is the string to be matched, and **value2** is the pattern string. The percent sign (%) matches any sequence of characters (including an empty sequence), and the underscore (_) matches exactly one character. To match a percent sign or underscore itself, use the **ESCAPE** clause to define an escape character. If any operand is **NULL**, **NULL** is returned.	SELECT 'hello' LIKE 'h%'; -- TRUE
value1 NOT LIKE value2	BOOLEAN	Returns **TRUE** if **value1** does not match pattern **value2**. The wildcard rules are the same as those for **LIKE**. This is the logical negation of **LIKE**. If any operand is **NULL**, **NULL** is returned.	SELECT 'hello' NOT LIKE 'a%'; -- TRUE
value1 IN (value2[, value3]*)	BOOLEAN	Determines whether a value is in a list. Returns **TRUE** if **value1** is equal to any value in the list. The type of **value1** must be compatible with the type of values in the list. If **value1** is not in the list and the list contains **NULL**, **NULL** is returned instead of **FALSE**. If **value1** is **NULL**, **NULL** is returned.	SELECT 2 IN (1, 2, 3); -- TRUE

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
value1 NOT IN (value2[, value3]*)	BOOLEAN	Determines whether a value is not in a list. If **value1** is not equal to any value in the list, **TRUE** is returned. This is the logical negation of **IN**. If the list contains **NULL** and **value1** is not in the list of non-null values, **NULL** is returned instead of **TRUE**. If value1 is **NULL**, **NULL** is returned.	SELECT 5 NOT IN (1, 2, 3); -- TRUE
value1 IS DISTINCT FROM value2	BOOLEAN	Determines whether values are different (NULL-safe). If the values of **value1** and **value2** are different, **TRUE** is returned. The key difference from **<>** is the NULL handling: **NULL IS DISTINCT FROM NULL** returns **FALSE**, and **NULL IS DISTINCT FROM** a non-null value returns **TRUE**. **NULL** results are not generated. This is used in scenarios where **NULLs** need to be treated as comparable values.	SELECT 1 IS DISTINCT FROM 2; -- TRUE
value1 IS NOT DISTINCT FROM value2	BOOLEAN	Determines whether values are the same (NULL-safe). If the values of **value1** and **value2** are the same, **TRUE** is returned. The key difference from **=** is the **NULL** handling: **NULL IS NOT DISTINCT FROM NULL** returns **TRUE**. **NULL** results are not generated. This is the logical negation of **IS DISTINCT FROM**. It is used in scenarios where **NULLs** need to be treated as equal values.	SELECT NULL IS NOT DISTINCT FROM NULL; -- TRUE

 Copyright © Huawei Technologies Co., Ltd.

6.2 Logical Functions

Overview

Logical functions perform logical operations on Boolean expressions and return results of the BOOLEAN type. AI DataLake supports three-valued logical operations (**AND**, **OR**, and **NOT**) in the SQL standard, **IS TRUE**/**IS FALSE** series Boolean judgment, and **IF** conditional functions, which are the basis for constructing complex query conditions.

For more information, see **Logical Functions**.

Application Scenarios

- **Combining filter conditions**: Uses **AND**/**OR** in the **WHERE** clause to combine multiple conditions.
- **Condition negation**: Uses **NOT** to negate a condition.
- **Boolean value determination**: Uses **IS TRUE**/**IS FALSE** to safely determine Boolean values, avoiding interference from **NULL**.
- **Condition-based value assignment**: Uses the **IF** function to return different expression values based on conditions.

Notes and Constraints

- **AND** and **OR** follow the three-valued logic of SQL. When any operand is **NULL**, the result may be **NULL**.
- The result of **NOT NULL** is still **NULL** and will not be changed to **TRUE** or **FALSE**.
- **IS TRUE**/**IS NOT TRUE**/**IS FALSE**/**IS NOT FALSE** always returns a definite **TRUE** or **FALSE** for **NULL**, and does not produce a **NULL** result.
- The types of the **true_value** and **false_value** parameters of the **IF** function must be compatible. Otherwise, explicit CAST conversion is required.

 Copyright © Huawei Technologies Co., Ltd.

Syntax Structure

Syntax	Return Type	Description	Example
boolean1 OR boolean2	BOOLEAN	Logical **OR**. If either operand is **TRUE**, the result is **TRUE**. If both operands are **FALSE**, the result is **FALSE**. When **NULL** is involved: if the other operand is **TRUE**, the result is **TRUE**; if the other operand is **FALSE**, the result is **NULL**. That is, **TRUE OR NULL** is **TRUE**, and **FALSE OR NULL** is **NULL**.	SELECT TRUE OR FALSE; -- TRUE
boolean1 AND boolean2	BOOLEAN	Logical **AND**. If both operands are **TRUE**, the result is **TRUE**. If either operand is **FALSE**, the result is **FALSE**. When **NULL** is involved: if the other operand is **FALSE**, the result is **FALSE**; if the other operand is **TRUE**, the result is **NULL**. That is, **FALSE AND NULL** is **FALSE**, and **TRUE AND NULL** is **NULL**.	SELECT TRUE AND FALSE; -- FALSE
NOT boolean	BOOLEAN	Logical **NOT**. It changes **TRUE** to **FALSE** and **FALSE** to **TRUE**. The result of **NOT NULL** is still **NULL**.	SELECT NOT TRUE; -- FALSE
boolean IS TRUE	BOOLEAN	Determines whether the value is **TRUE**. Returns **TRUE** if the boolean is **TRUE**, returns **FALSE** if the boolean is **FALSE**, and returns **FALSE** if the boolean is **NULL**. Unlike using boolean directly, **IS TRUE** returns a definite **FALSE** instead of **NULL** for **NULL**.	SELECT NULL IS TRUE; -- FALSE

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
boolean IS NOT TRUE	BOOLEAN	Determines whether the value is not **TRUE**. Returns **FALSE** if the boolean is **TRUE**, and returns **TRUE** if the boolean is **FALSE** or **NULL**. This is equivalent to **NOT(boolean IS TRUE)**, but the result is always a definite **TRUE** or **FALSE**.	SELECT NULL IS NOT TRUE; -- TRUE
boolean IS FALSE	BOOLEAN	Determines whether the value is **FALSE**. Returns **TRUE** if the boolean is **FALSE**, returns **FALSE** if the boolean is **TRUE**, and returns **FALSE** if the boolean is **NULL**. Returns a definite **FALSE** instead of **NULL** for **NULL**.	SELECT NULL IS FALSE; -- FALSE
boolean IS NOT FALSE	BOOLEAN	Determines whether the value is not **FALSE**. Returns **FALSE** if the boolean is **FALSE**, and returns **TRUE** if the boolean is **TRUE** or **NULL**. This is equivalent to **NOT(boolean IS FALSE)**, but the result is always a definite **TRUE** or **FALSE**.	SELECT NULL IS NOT FALSE; -- TRUE
IF(condition, true_value, false_value)	Consistent with **true_value**/ **false_value**	Conditional function. If **condition** is of the BOOLEAN type, **true_value** is returned if **condition** is **TRUE**, and **false_value** is returned if **condition** is **FALSE** or **NULL**. The types of **true_value** and **false_value** must be compatible.	SELECT IF(3 > 2, 'yes', 'no'); -- 'yes'

 Copyright © Huawei Technologies Co., Ltd.

6.3 Arithmetic Functions

Overview

Arithmetic functions perform mathematical operations on numeric types, including unary positive and negative signs, four arithmetic operations, modulo, exponentiation, square root, logarithm, exponent, trigonometric functions, rounding and truncation, sign determination, mathematical constants, random numbers, and base conversion.

For more information, see **Arithmetic Functions**.

Application Scenarios

- **Numeric calculation**: Performs addition, subtraction, multiplication, and division on columns in the **SELECT** statement to calculate service metrics.
- **Precision control**: Uses **CEIL**, **FLOOR**, **ROUND**, and **TRUNCATE** to convert and truncate the numeric precision.
- **Scientific calculation**: Performs scientific and engineering calculations using trigonometric functions, logarithms, and exponential functions.
- **Data conversion**: Converts values between number systems and strings using **BIN** and **HEX**.
- **Random sampling**: Uses **RAND** to generate random numbers for data sampling or test data construction.

Notes and Constraints

- Integer division truncates the fractional part. To retain the fractional part, use the DECIMAL type or divide the value by a DECIMAL value.
- Division by zero produces a NULL result.
- The input and return types of **POWER** and **SQRT** are **DOUBLE**.
- The input of **LN**, **LOG10**, and **LOG2** must be greater than 0. Otherwise, **NULL** is returned.
- The parameters of the trigonometric functions SIN, COS, TAN, and COT are radians instead of degrees. The input ranges of the inverse trigonometric functions ASIN and ACOS are [-1, 1]. If the input is out of the range, **NULL** is returned.
- If any input of an arithmetic function is **NULL**, the result is **NULL**.
- **TRUNCATE** directly truncates the value instead of rounding it, which is different from the rounding behavior of **ROUND**.

 Copyright © Huawei Technologies Co., Ltd.

Syntax Structure

Syntax	Return Type	Description	Example
+ numeric	Same as the input	Unary plus sign. Returns the value of numeric itself without changing the value. The numeric type can be **TINYINT**, **SMALLINT**, **INT**, **BIGINT**, **FLOAT**, **DOUBLE**, or **DECIMAL**.	`SELECT +5; -- 5`
- numeric	Same as the input	Unary minus sign. Returns the negation of numeric. The numeric type can be any numeric type.	`SELECT -5; -- -5`
numeric1 + numeric2	Automatic type promotion	Addition operation. Returns the sum of **numeric1** and **numeric2**. The two operands are of any numeric type, and the result type is automatically promoted according to the SQL standard (for example, INT + DOUBLE → DOUBLE).	`SELECT 10 + 5; -- 15`
numeric1 - numeric2	Automatic type promotion	Subtraction operation. Returns the difference between **numeric1** and **numeric2**. The two operands are of any numeric type, and the result type is automatically promoted.	`SELECT 10 - 5; -- 5`
numeric1 * numeric2	Automatic type promotion	Multiplication operation. Returns the product of **numeric1** and **numeric2**. The two operands are of any numeric type, and the result type is automatically promoted.	`SELECT 10 * 5; -- 50`
numeric1 / numeric2	Automatic type promotion	Division operation. Returns the quotient of **numeric1** divided by **numeric2**. For integer division, the fractional part is truncated. To retain the fractional part, use the DECIMAL type. If the divisor is zero, **NULL** is returned.	`SELECT 10 / 3; -- 3`

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
POWER(numeric1, numeric2)	DOUBLE	Exponentiation operation. Returns the numeric2th power of **numeric1**, that is, **numeric1^numeric2**. If both parameters are of the numeric type, **DOUBLE** is returned. If **numeric1** is a negative number and **numeric2** is not an integer, the result is **NULL**.	SELECT POWER(2, 3); -- 8.0
ABS(numeric)	Same as the input	Absolute value. Returns the absolute value of **numeric**. If **numeric** is of any numeric type, the return type is the same as the input.	SELECT ABS(-5); -- 5
MOD(numeric1, numeric2)	Same as the input	Modulo operation. Returns the remainder of **numeric1** divided by **numeric2**, which is equivalent to the % operator. If **numeric2** is 0, **NULL** is returned. If both parameters are of the integer type, the return value is an integer. Otherwise, the return value is of the DECIMAL type.	SELECT MOD(10, 3); -- 1
SQRT(numeric)	DOUBLE	Square root. Returns the non-negative square root of **numeric**. If **numeric** is a negative number, **NULL** is returned.	SELECT SQRT(16); -- 4.0
LN(numeric)	DOUBLE	Natural logarithm. Returns the natural logarithm of **numeric** (base e). The value of **numeric** must be greater than 0. Otherwise, **NULL** is returned.	SELECT LN(2.718281828); -- 0.9999999995
LOG10(numeric)	DOUBLE	Base-10 logarithm. Returns the base-10 logarithm of **numeric**. The value of **numeric** must be greater than 0. Otherwise, **NULL** is returned.	SELECT LOG10(100); -- 2.0

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
LOG2(numeric)	DOUBLE	Base-2 logarithm. Returns the base-2 logarithm of **numeric**. The value of **numeric** must be greater than 0. Otherwise, **NULL** is returned.	SELECT LOG2(8); -- 3.0
LOG(numeric1, numeric2)	DOUBLE	Custom-base logarithm. Returns the logarithm of **numeric1** with **numeric2** as the base. The value of **numeric1** must be greater than 0, and the value of **numeric2** must be greater than 0 and not equal to 1. Otherwise, **NULL** is returned.	SELECT LOG(8, 2); -- 3.0
EXP(numeric)	DOUBLE	Exponential operation. Returns the **numeric** power of e (the natural constant), that is, e^numeric. It is the inverse function of LN.	SELECT EXP(1); -- 2.718281828459045
CEIL(numeric) / CEILING(numeric)	Same as the input	Ceiling. Returns the smallest integer greater than or equal to **numeric**. The **numeric** value can be of any numeric type.	SELECT CEIL(3.2); -- 4
FLOOR(numeric)	Same as the input	Floor. Returns the largest integer less than or equal to **numeric**. The **numeric** value can be of any numeric type.	SELECT FLOOR(3.8); -- 3
ROUND(numeric, integer)	Same as the input	Rounding. Rounds **numeric** to **integer** decimal places. If **integer** is positive, the specified number of decimal places is retained. If **integer** is negative, the integer part is rounded. If **integer** is 0, the number is rounded to the nearest integer.	SELECT ROUND(3.456, 2); -- 3.46

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
TRUNCATE(numeric, integer)	Same as the input	Truncation. Truncates the **numeric** value to the specified number of **integer** decimal places. The redundant decimal places are discarded without rounding. If the integer part is positive, the corresponding decimal places are retained. If the integer part is negative, the integer part is truncated. If the integer part is 0, the value is truncated to an integer.	SELECT TRUNCATE(3.456, 2); -- 3.45
SIGN(numeric)	INT	Sign function. Returns the sign of a **numeric** value. If the **numeric** value is greater than 0, **1** is returned. If the **numeric** value is equal to 0, **0** is returned. If the **numeric** value is less than 0, **-1** is returned.	SELECT SIGN(-5); -- -1
SIN(numeric)	DOUBLE	Sine function. Returns the sine value of **numeric** (in radians), where **numeric** is a numeric type indicating radians.	SELECT SIN(0); -- 0.0
COS(numeric)	DOUBLE	Cosine function. Returns the cosine value of **numeric** (in radians), where **numeric** is a numeric type indicating radians.	SELECT COS(0); -- 1.0
TAN(numeric)	DOUBLE	Tangent function. Returns the tangent value of **numeric** (in radians), where **numeric** is a numeric type indicating radians.	SELECT TAN(0); -- 0.0
COT(numeric)	DOUBLE	Cotangent function. Returns the cotangent value of **numeric** (in radians), that is, 1/TAN(numeric). If **numeric** is 0 or an integer multiple of π, **NULL** is returned.	SELECT COT(1); -- 0.6420926159343308

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
ASIN(numeric)	DOUBLE	Arcsine function. Returns the arcsine value of **numeric** (in radians). The value of **numeric** must be in the range of [-1, 1]. If the value is out of the range, **NULL** is returned.	SELECT ASIN(1); -- 1.5707963267948966
ACOS(numeric)	DOUBLE	Arc cosine function. Returns the arc cosine value of **numeric** (in radians). The value of **numeric** must be in the range of [-1, 1]. If the value is out of the range, **NULL** is returned.	SELECT ACOS(1); -- 0.0
ATAN(numeric)	DOUBLE	Arc tangent function. Returns the arc tangent of **numeric** (in radians). The **numeric** value can be of any numeric type, and the return value range is (-π/2, π/2).	SELECT ATAN(1); -- 0.7853981633974483
ATAN2(numeric1, numeric2)	DOUBLE	Two-argument arc tangent function. Returns the angle (in radians) between the point (**numeric2**, **numeric1**) and the positive x-axis. Unlike **ATAN**, **ATAN2** determines the quadrant based on the two arguments, and the return value range is (-π, π].	SELECT ATAN2(1, 1); -- 0.7853981633974483
PI()	DOUBLE	Pi function. Returns the approximate value of π, which is 3.141592653589793. This function takes no parameters.	SELECT PI(); -- 3.141592653589793
E()	DOUBLE	Natural constant. Returns the approximate value of e, 2.718281828459045. No parameter is required.	SELECT E(); -- 2.718281828459045
RAND()	DOUBLE	Random number. Returns a pseudo-random DOUBLE value in the range of [0.0, 1.0). No parameter is required. Each time the function is called, a different value is returned.	SELECT RAND(); -- 0.123... (random value)

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
RAND(integer)	DOUBLE	Seed random number. Returns a pseudo-random DOUBLE value in the range of [0.0, 1.0). The **integer** is the seed value. The same seed generates the same random sequence.	SELECT RAND(1); -- 0.730... (fixed seed)
BIN(integer)	STRING	Converts an integer to a binary string. Returns the binary representation of the **integer**. The **integer** is of an integer type (TINYINT, SMALLINT, INT, or BIGINT). If the **integer** is a negative number, the binary representation of its two's complement is returned.	SELECT BIN(5); -- '101'
HEX(numeric)	STRING	Converts a value to a hexadecimal string. Returns the hexadecimal representation of a **numeric** value. If the **numeric** value is of the integer type, its hexadecimal representation is returned. If the **numeric** value is of the STRING type, each character is converted to a two-digit hexadecimal number.	SELECT HEX(255); -- 'FF'
BITAND	INT	The BITAND function performs a bitwise AND operation. The bitwise AND operation compares the binary bits of two integers bit by bit. If both corresponding bits are **1**, the result bit is **1**. Otherwise, the result bit is **0**.	For details, see **8.1.1 BITAND**.

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
BITNOT	INT	The BITNOT function performs a bitwise NOT operation. The bitwise NOT operation inverts the binary bits of an integer, that is, 0 becomes 1 and 1 becomes 0. Because the INT type uses two's complement representation, the result after inversion must be understood according to the two's complement rules.	For details, see **8.1.2 BITNOT**.
BITOR	INT	The BITOR function is used to perform the bitwise OR operation. The bitwise OR operation compares the binary bits of two integers bit by bit. If at least one of the two corresponding bits is **1**, the result bit is **1**. Otherwise, the result bit is **0**.	For details, see **8.1.3 BITOR**.
BITXOR	INT	The BITXOR function is used to perform the bitwise XOR operation. The bitwise XOR operation compares the binary bits of two integers bit by bit. If the two corresponding bits are different, the result bit is **1**. If they are the same, the result bit is **0**.	For details, see **8.1.4 BITXOR**.

6.4 String Functions

Overview

String functions are used to perform operations on data of the VARCHAR or STRING type, including concatenation, truncation, replacement, case conversion, trimming, length calculation, padding, and pattern matching.

For more information, see **String Functions**.

Application Scenarios

- **Data cleansing**: Uses **TRIM** to remove spaces and use **UPPER** or **LOWER** to convert the case to uppercase or lowercase.

 Copyright © Huawei Technologies Co., Ltd.

- **String concatenation**: Uses **CONCAT** or **CONCAT_WS** to concatenate multiple fields.
- **Data masking**: Uses **SUBSTRING** and **REPLACE** to mask fields.
- **Pattern replacement**: Uses **REGEXP_REPLACE** to perform regular expression replacement.
- **Formatted output**: Uses **LPAD** or **RPAD** to align fields and uses **FORMAT_NUMBER** to format numbers.
- **Pattern matching**: Uses **LIKE** for simple pattern matching and **REGEXP** for regular expression matching.

Notes and Constraints

- If any parameter in **CONCAT** is **NULL**, the result is **NULL**. **CONCAT_WS** skips **NULL** parameters.
- The start position of **SUBSTRING** is **1**. The value 0 is equivalent to 1.
- **REGEXP_REPLACE** uses the Java regular expression syntax.
- When **string1** is specified in the **TRIM** function, only single characters that exactly match **string1** are removed. This is not a substring deletion operation.
- If **len** is less than the original length of the **string**, **LPAD** or **RPAD** truncates the string to the length of **len**.

Syntax Structure

Syntax	Return Type	Description	Example
string1 || string2	STRING	String concatenation. Concatenates two strings. The **string1** and **string2** are concatenated end-to-end to form a new string. Both **string1** and **string2** must be of the STRING type. If either operand is **NULL**, **NULL** is returned.	`SELECT 'hello' || ' ' || 'world'; -- 'hello world'`
CHAR_LENGTH(string)	INT	Returns the number of characters in a string. The **string** must be of the STRING type. This function is equivalent to **CHARACTER_LENGTH**. A multi-byte character (such as Chinese) is counted as one character. If the **string** is **NULL**, **NULL** is returned.	`SELECT CHAR_LENGTH('hello'); -- 5`

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
CHARACTER_LENGTH(string)	INT	Returns the number of characters in a string. This function is equivalent to **CHAR_LENGTH**, which is a SQL standard alias. The **string** must be of the STRING type. If the **string** is **NULL**, **NULL** is returned.	SELECT CHARACTER_LENGTH('Hello'); -- 2
UPPER(string)	STRING	Converts a string to uppercase. All lowercase letters in the string are converted to uppercase letters, and non-letter characters remain unchanged. The **string** must be of the STRING type. If the **string** is **NULL**, **NULL** is returned.	SELECT UPPER('hello'); -- 'HELLO'
LOWER(string)	STRING	Converts a string to lowercase. All uppercase letters in the string are converted to lowercase letters, and non-letter characters remain unchanged. The **string** must be of the STRING type. If the **string** is **NULL**, **NULL** is returned.	SELECT LOWER('HELLO'); -- 'hello'
INITCAP(string)	STRING	Converts the first letter of each word in a string to uppercase and the other letters to lowercase. Words are separated by non-letter characters. The **string** must be of the STRING type. If the **string** is **NULL**, **NULL** is returned.	SELECT INITCAP('hello world'); -- 'Hello World'

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
TRIM([BOTH | LEADING | TRAILING] string1 FROM string2)	STRING	Removes characters that match **string1** from the beginning (**LEADING**), end (**TRAILING**), or both ends (**BOTH**, default) of **string2**. If **string1** and **BOTH** are omitted, spaces at both ends are removed by default. Both **string1** and **string2** must be of the STRING type. If any parameter is **NULL**, **NULL** is returned.	SELECT TRIM(BOTH 'x' FROM 'xxhelloxx'); -- 'hello'
LTRIM(string)	STRING	Removes leading spaces from the left side of a string. The **string** must be of the STRING type. If the **string** is **NULL**, **NULL** is returned.	SELECT LTRIM(' hello'); -- 'hello'
RTRIM(string)	STRING	Removes trailing spaces. All space characters on the right side of the **string** are removed. The **string** must be of the STRING type. If the string is **NULL**, **NULL** is returned.	SELECT RTRIM('hello '); -- 'hello'

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
SUBSTRING(string FROM pos [FOR len])	STRING	Extracts a substring (SQL standard syntax). A substring of the specified length is extracted from the **string** starting at the position specified by **pos**. The value of **pos** starts from **1**. The value **0** is equivalent to **1**. If **FOR len** is omitted, the substring is extracted to the end of the **string**. If **pos** is a negative number, the substring is extracted from the end of the **string**. The **string** must be of the STRING type, and **pos** must be of the INT type. If the **string** is **NULL**, **NULL** is returned.	SELECT SUBSTRING('hello world' FROM 1 FOR 5); -- 'hello'
SUBSTRING(string, pos [, len])	STRING	Extracts a substring (comma syntax). This function is semantically equivalent to **SUBSTRING(string FROM pos [FOR len])**, but the syntax is different. The value of **pos** starts from **1**. The value **0** is equivalent to **1**. If **len** is omitted, the substring is extracted to the end of the **string**. The **string** must be of the STRING type, and **pos** must be of the INT type. If the **string** is **NULL**, **NULL** is returned.	SELECT SUBSTRING('hello world', 7); -- 'world'

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
CONCAT(string1, string2, ...)	STRING	Concatenates strings. All parameters are concatenated into a single string in sequence. All parameters must be of the STRING type. Two or more parameters are supported. The key difference from the **||** operator is that if any parameter is **NULL**, the entire result is **NULL**.	SELECT CONCAT('hello', ' ', 'world'); -- 'hello world'
CONCAT_WS(separator, string1, ...)	STRING	Concatenates all non-null string parameters with a separator. The **separator** and **string** parameters must be of the STRING type. The key difference from **CONCAT** is that **NULL** parameters are skipped instead of being returned. If the **separator** is **NULL**, **NULL** is returned.	SELECT CONCAT_WS('-', '2024', '01', '15'); -- '2024-01-15'
REPLACE(string, old, new)	STRING	Replaces all occurrences of the old substring in **string** with the new substring. The replacement is case-sensitive. If the **old** substring is an empty string, the replacement is not performed. The **string**, **old**, and **new** parameters must be of the STRING type. If any parameter is **NULL**, **NULL** is returned.	SELECT REPLACE('hello world', 'world', 'Flink'); -- 'hello Flink'

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
REGEXP_REPLACE(string, regex, replacement)	STRING	Replaces all substrings that match the regular expression in **string** with the **replacement**. The Java regular expression syntax is used. The **replacement** can use 1, 1 and 2, and so on to reference capture groups. The **string**, **regex**, and **replacement** parameters must be of the STRING type. If any parameter is **NULL**, **NULL** is returned.	SELECT REGEXP_REPLACE('abc123', '[0-9]', '*'); -- 'abc***'
REGEXP_EXTRACT(string, regex [, group])	STRING	Extracts a substring that matches the regular expression from a **string**. **group** specifies the ID of the capture group to be returned. The default value is **0**, indicating that the entire match is returned. **string** and **regex** must be of the STRING type, and **group** must be of the INT type. If no match is found, **NULL** is returned. If any parameter is **NULL**, **NULL** is returned.	SELECT REGEXP_EXTRACT('abc123def', '([a-z]+)([0-9]+)', 2); -- '123'
POSITION(string1 IN string2)	INT	Returns the starting position where **string1** first appears in **string2**, starting from 1. If **string1** is not found, **0** is returned. **string1** and **string2** must be of the STRING type. If any parameter is **NULL**, **NULL** is returned.	SELECT POSITION('world' IN 'hello world'); -- 7

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
LPAD(string, len, pad)	STRING	Pads the left side of a **string** with the **pad** string until the total length reaches **len**. If the length of the **string** exceeds **len**, the **string** is truncated to **len** from the right. **string** and **pad** must be of the STRING type, and **len** must be of the INT type. If any parameter is **NULL**, **NULL** is returned.	`SELECT LPAD('5', 3, '0'); -- '005'`
RPAD(string, len, pad)	STRING	Pads the right side of a **string** with the **pad** string until the total length reaches **len**. If the length of the **string** exceeds **len**, the **string** is truncated to **len** from the right. **string** and **pad** must be of the STRING type, and **len** must be of the INT type. If any parameter is **NULL**, **NULL** is returned.	`SELECT RPAD('5', 3, '0'); -- '500'`
OVERLAY(string1 PLACING string2 FROM pos [FOR len])	STRING	Overwrites a substring. It overwrites **len** characters in **string1** starting from position **pos**. The value of **pos** starts from **1**. If **FOR len** is omitted, the length of the string to be overwritten is the length of **string2**. **string1** and **string2** must be of the STRING type, and **pos** and **len** must be of the INT type. If any parameter is **NULL**, **NULL** is returned.	`SELECT OVERLAY('hello world' PLACING 'FLINK' FROM 7); -- 'hello FLINK'`
REVERSE(string)	STRING	Reverses a string. The order of characters in **string** is reversed. **string** must be of the STRING type. If **string** is **NULL**, **NULL** is returned.	`SELECT REVERSE('hello'); -- 'olleh'`

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
REPEAT(string, integer)	STRING	Repeats a string. The **string** is repeated **integer** times and then concatenated into a new **string**. **string** must be of the STRING type, and **integer** must be of the INT type. If **integer** is **0**, an empty string is returned. If **integer** is a negative number, **NULL** is returned. If **string** is **NULL**, **NULL** is returned.	`SELECT REPEAT('ab', 3); -- 'ababab'`
SPACE(integer)	STRING	Generates a string of spaces. A string consisting of **integer** spaces is returned. **integer** must be of the INT type. If **integer** is **0**, an empty string is returned. If **integer** is a negative number, **NULL** is returned.	`SELECT SPACE(3); -- '  '`
LEFT(string, integer)	STRING	Left truncation. Returns the leftmost **integer** characters of a **string**. The **string** must be of the STRING type, and the **integer** must be of the INT type. If the **integer** is **0**, an empty string is returned. If the **integer** is a negative number, **NULL** is returned. If the **string** is **NULL**, **NULL** is returned.	`SELECT LEFT('hello world', 5); -- 'hello'`

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
RIGHT(string, integer)	STRING	Right truncation. Returns the rightmost **integer** characters of a **string**. The **string** must be of the STRING type, and the **integer** must be of the INT type. If the **integer** is **0**, an empty string is returned. If the **integer** is a negative number, **NULL** is returned. If the **string** is **NULL**, **NULL** is returned.	SELECT RIGHT('hello world', 5); -- 'world'
FORMAT_NUMBER(numeric, decimal_places)	STRING	Formats a number. The **numeric** value is formatted into a string with thousands separators, and **decimal_places** decimal places are retained. The **numeric** value must be of a numeric type, and **decimal_places** must be of the INT type. If either parameter is **NULL**, **NULL** is returned.	SELECT FORMAT_NUMBER(12345.6789, 2); -- '12,345.68'
LIKE(string1, string2)	BOOLEAN	Pattern matching. Determines whether **string1** matches the pattern **string2**. The percent sign (%) matches any sequence of characters (including an empty sequence), and the underscore (_) matches exactly one character. To match a wildcard character itself, use the **ESCAPE** clause for escaping. Both **string1** and **string2** must be of the STRING type. If either parameter is **NULL**, **NULL** is returned.	SELECT LIKE('hello', 'h%'); -- TRUE

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
REGEXP(string1, string2)	BOOLEAN	Performs regular expression matching. Determines whether **string1** matches the regular expression **string2**. The Java regular expression syntax is used. Both **string1** and **string2** must be of the STRING type. If either parameter is **NULL**, **NULL** is returned.	SELECT REGEXP('abc123', '[a-z]+[0-9]+'); -- TRUE
URL_DECODE	VARCHAR	The **URL_DECODE** function is used to convert an input string in **application/x-www-form-urlencoded MIME** format to a standard string.	For details, see **8.1.8 URL_DECODE**.
URL_ENCODE	VARCHAR	The **URL_ENCODE** function is used to encode an input string in **application/x-www-form-urlencoded MIME** format and return the encoded string.	For details, see **8.1.9 URL_ENCODE**.

6.5 Time Functions

Overview

Time functions are used to perform operations on data of the DATE, TIME, and TIMESTAMP types, including type conversion, obtaining the current time, extracting time components, formatting, rounding time, and calculating time offsets.

For more information, see **Temporal Functions**.

Application Scenarios

- **Watermark definition**: The **CURRENT_TIMESTAMP** and **INTERVAL** functions are used to define the watermark policy.
- **Window grouping**: The **DATE_FORMAT** function is used to format the time into hour- or day-level for window grouping.

 Copyright © Huawei Technologies Co., Ltd.

- **Time calculation**: The **TIMESTAMPADD** and **TIMESTAMPDIFF** functions are used to calculate the time offset and difference.
- **ETL time field processing**: The year, month, and day components are extracted for data partitioning and aggregation.

Notes and Constraints

- **CURRENT_TIMESTAMP** returns the timestamp in the UTC time zone, and **LOCALTIMESTAMP** returns the timestamp in the local time zone.
- **DATE_FORMAT** uses the Java DateTimeFormatter format pattern (such as **yyyy-MM-dd** and **HH:mm:ss**), which is different from the SimpleDateFormat pattern.
- **INTERVAL** supports only the year and month interval (**YEAR** and **MONTH**) and the day, hour, minute, and second interval (**DAY**, **HOUR**, **MINUTE**, and **SECOND**). These intervals cannot be used together.
- **PROCTIME()** can be used only to declare the processing time attribute in DDL statements and cannot be directly called in queries.

Syntax Structure of Type Conversion

Syntax	Return Type	Description	Example
DATE string	DATE	Converts a string to the DATE type. The **string** parameter must be of the DATE or STRING type and in the format of **yyyy-MM-dd**. Otherwise, **NULL** is returned. The return value is of the DATE type.	`SELECT DATE '2024-01-15'; -- 2024-01-15`
TIME string	TIME	Converts a string to the TIME type. The **string** parameter must be of the TIME or STRING type and in the format of **HH:mm:ss**. Otherwise, **NULL** is returned. The return value is of the TIME type.	`SELECT TIME '10:30:00'; -- 10:30:00`
TIMESTAMP string	TIMESTAMP	Converts a string to the TIMESTAMP type. The **string** parameter must be of the TIMESTAMP or STRING type and in the format of **yyyy-MM-dd HH:mm:ss[.SSSSSS]**. Otherwise, **NULL** is returned. The return value is of the TIMESTAMP type.	`SELECT TIMESTAMP '2024-01-15 10:30:00'; -- 2024-01-15 10:30:00`

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
INTERVAL string range	INTERVAL	Converts a string to an interval. The **string** parameter is of the STRING type and indicates the interval. The **range** parameter is a unit qualifier, which can be **YEAR**, **MONTH**, **DAY**, **HOUR**, **MINUTE**, **SECOND**, **YEAR TO MONTH**, or **DAY TO SECOND**. The return value is of the INTERVAL type.	SELECT INTERVAL '1' DAY; -- +1 00:00:00.000
TO_DATE(string[, format])	DATE	Converts a string to the DATE type in the specified format. The **string** parameter is of the STRING type. The **format** parameter is an optional Java DateTimeFormatter string, which defaults to **yyyy-MM-dd**. If the format does not match, **NULL** is returned. The return value is of the DATE type.	SELECT TO_DATE('2024-01-15', 'yyyy-MM-dd'); -- 2024-01-15
TO_TIMESTAMP(string[, format])	TIMESTAMP	Converts a string to the TIMESTAMP type in the specified format. The **string** parameter is of the STRING type. The **format** parameter is an optional Java DateTimeFormatter string, which defaults to **yyyy-MM-dd HH:mm:ss**. If the format does not match, **NULL** is returned. The return value is of the TIMESTAMP type.	SELECT TO_TIMESTAMP('2024-01-15 10:30:00', 'yyyy-MM-dd HH:mm:ss'); -- 2024-01-15 10:30:00

 Copyright © Huawei Technologies Co., Ltd.

Syntax for Obtaining Current Time Class

Syntax	Return Type	Description	Example
CURRENT_DATE	DATE	Returns the date of the current SQL session in the UTC time zone. It takes no parameters. The return value is of the DATE type.	SELECT CURRENT_DATE; -- 2024-01-15
CURRENT_TIME	TIME WITH TIME ZONE	Returns the time of the current SQL session with time zone. It takes no parameters. The return value is of the TIME WITH TIME ZONE type.	SELECT CURRENT_TIME; -- 10:30:00+08:00
CURRENT_TIMESTAMP	TIMESTAMP_LTZ(3)	Returns the timestamp of the current SQL session with the local time zone and a precision of three milliseconds. It takes no parameters. The return value is of the TIMESTAMP_LTZ(3) type.	SELECT CURRENT_TIMESTAMP; -- 2024-01-15T10:30:00.000+08:00
LOCALTIME	TIME	Returns the local time of the current SQL session (without time zone). It takes no parameters. The return value is of the TIME type.	SELECT LOCALTIME; -- 10:30:00
LOCALTIMESTAMP	TIMESTAMP(3)	Returns the local timestamp of the current SQL session (without time zone, with millisecond precision of three digits). It takes no parameters. The return value is of the TIMESTAMP(3) type.	SELECT LOCALTIMESTAMP; -- 2024-01-15 10:30:00.000
NOW()	TIMESTAMP_LTZ(3)	Returns the current timestamp, which is equivalent to **CURRENT_TIMESTAMP**. It takes no parameters. The return value is of the TIMESTAMP_LTZ(3) type.	SELECT NOW(); -- 2024-01-15T10:30:00.000+08:00

 Copyright © Huawei Technologies Co., Ltd.

Syntax for Extracting Time Components

Syntax	Return Type	Description	Example
EXTRACT(timeintervalunit FROM temporal)	INT	Extracts the value of the specified time part from a time value. The **timeintervalunit** parameter is a time unit enumeration, which can be **YEAR**, **MONTH**, **DAY**, **HOUR**, **MINUTE**, **SECOND**, **QUARTER**, **WEEK**, **DOW**, or **DOY**. The **temporal** parameter must be of the DATE, TIME, TIMESTAMP, or INTERVAL type. The return value is of the INT type.	SELECT EXTRACT(YEAR FROM TIMESTAMP '2024-01-15 10:30:00'); -- 2024
YEAR(date)	INT	Extracts the year from a date. The **date** parameter must be of the DATE or TIMESTAMP type. The return value is of the INT type, ranging from 1 to 9999.	SELECT YEAR(DATE '2024-01-15'); -- 2024
QUARTER(date)	INT	Extracts the quarter from a date. The **date** parameter must be of the DATE or TIMESTAMP type. The return value is of the INT type, ranging from 1 to 4.	SELECT QUARTER(DATE '2024-04-15'); -- 2
MONTH(date)	INT	Extracts the month from a date. The **date** parameter must be of the DATE or TIMESTAMP type. The return value is of the INT type, ranging from 1 to 12.	SELECT MONTH(DATE '2024-01-15'); -- 1
WEEK(date)	INT	Extracts the week number of the year from a date. The **date** parameter must be of the DATE or TIMESTAMP type. The return value is of the INT type, ranging from 1 to 53.	SELECT WEEK(DATE '2024-01-15'); -- 3
DAYOFYEAR(date)	INT	Extracts the day number of the year from a date. The **date** parameter must be of the DATE or TIMESTAMP type. The return value is of the INT type, ranging from 1 to 366.	SELECT DAYOFYEAR(DATE '2024-02-01'); -- 32

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
DAYOFMONTH(date)	INT	Extracts the day number of the month from a date. The **date** parameter must be of the DATE or TIMESTAMP type. The return value is of the INT type, ranging from 1 to 31.	SELECT DAYOFMONTH(DATE '2024-01-15'); -- 15
DAYOFWEEK(date)	INT	Extracts the day of the week from a date. The **date** parameter must be of the DATE or TIMESTAMP type. The return value is of the INT type, ranging from 1 (Sunday) to 7 (Saturday).	SELECT DAYOFWEEK(DATE '2024-01-15'); -- 2
HOUR(timestamp)	INT	Extracts the hour from the timestamp. The **timestamp** parameter must be of the TIME or TIMESTAMP type. The return value is of the INT type, ranging from 0 to 23.	SELECT HOUR(TIMESTAMP '2024-01-15 10:30:00'); -- 10
MINUTE(timestamp)	INT	Extracts the minute from the timestamp. The **timestamp** parameter must be of the TIME or TIMESTAMP type. The return value is of the INT type, ranging from 0 to 59.	SELECT MINUTE(TIMESTAMP '2024-01-15 10:30:00'); -- 30
SECOND(timestamp)	INT	Extracts the second from the timestamp. The **timestamp** parameter must be of the TIME or TIMESTAMP type. The return value is of the INT type, ranging from 0 to 59.	SELECT SECOND(TIMESTAMP '2024-01-15 10:30:45'); -- 45

 Copyright © Huawei Technologies Co., Ltd.

Syntax for Rounding Time

Syntax	Return Type	Description	Example
FLOOR(temporal TO timeintervalunit)	Consistent with the input type	Rounds down the time value to the specified time unit. The **temporal** parameter must be of the DATE, TIME, or TIMESTAMP type. The **timeintervalunit** parameter indicates the rounding unit, which can be **YEAR**, **MONTH**, **DAY**, **HOUR**, **MINUTE**, or **SECOND**. The return value type is the same as the input type.	SELECT FLOOR(TIMESTAMP '2024-01-15 10:30:45' TO HOUR); -- 2024-01-15 10:00:00
CEIL(temporal TO timeintervalunit)	Consistent with the input type	Rounds up the time value to the specified time unit. The **temporal** parameter must be of the DATE, TIME, or TIMESTAMP type. The **timeintervalunit** parameter indicates the rounding unit, which can be **YEAR**, **MONTH**, **DAY**, **HOUR**, **MINUTE**, or **SECOND**. The return value type is the same as the input type.	SELECT CEIL(TIMESTAMP '2024-01-15 10:30:45' TO HOUR); -- 2024-01-15 11:00:00

Time Formatting Class

Syntax	Return Type	Description	Example
DATE_FORMAT(timestamp, format)	STRING	Converts a **timestamp** to a string in the specified **format**. The **timestamp** parameter must be of the TIMESTAMP or TIMESTAMP_LTZ type. The **format** parameter is of the STRING type and uses the Java DateTimeFormatter format pattern (such as **yyyy-MM-dd**, **HH:mm:ss**, and **yyyyMMdd**). The return value is of the STRING type.	SELECT DATE_FORMAT(TIMESTAMP '2024-01-15 10:30:00', 'yyyy-MM-dd'); -- '2024-01-15'

 Copyright © Huawei Technologies Co., Ltd.

Time Calculation Class

Syntax	Return Type	Description	Example
TIMESTAMPADD(unit, interval, timestamp)	TIMESTAMP	Adds a specified **interval** to a **timestamp**. The **unit** parameter is the time unit, which can be **YEAR**, **MONTH**, **QUARTER**, **WEEK**, **DAY**, **HOUR**, **MINUTE**, or **SECOND**. The **interval** parameter must be of the INT type, indicating the number of intervals to be added (a negative number can be used to implement subtraction). The **timestamp** parameter must be of the TIMESTAMP type. The return value is of the TIMESTAMP type.	SELECT TIMESTAMPADD(DAY, 7, TIMESTAMP '2024-01-15 10:30:00'); -- 2024-01-22 10:30:00
TIMESTAMPDIFF(unit, timestamp1, timestamp2)	INT	Calculates the difference between two timestamps. The **unit** parameter specifies the time unit, which can be **YEAR**, **MONTH**, **QUARTER**, **WEEK**, **DAY**, **HOUR**, **MINUTE**, or **SECOND**. The **timestamp1** and **timestamp2** parameters must be of the TIMESTAMP type. The return value is of the INT type, indicating the difference between **timestamp2** and **timestamp1** in the specified **unit**.	SELECT TIMESTAMPDIFF(DAY, TIMESTAMP '2024-01-01 00:00:00', TIMESTAMP '2024-01-15 00:00:00'); -- 14
DATE_ADD	STRING	The **DATE_ADD** function returns the date after adding a specified number of days to a given date. If the target number of days is negative, the corresponding number of days is subtracted.	For details, see **8.1.5 DATE_ADD**.
DATE_SUB	STRING	The **DATE_SUB** function returns the date after subtracting a specified number of days from a given date. If the target number of days is negative, the corresponding number of days is added.	For details, see **8.1.6 DATE_SUB**.

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
DATEDIFF	INT	The **DATEDIFF** function calculates the day difference between two dates. The calculation subtracts **startdate** from **enddate**, meaning the return value equals **enddate** – **startdate**. A positive number is returned if **enddate** is later than **startdate**, and a negative number is returned if **enddate** is earlier than **startdate**.	For details, see **8.1.7 DATEDIFF**.

Time Attribute Processing Class

Syntax	Return Type	Description	Example
PROCTIME()	TIMESTAMP_LTZ(3)	Declares the processing time attribute and returns the current processing timestamp. It takes no parameters. It can be used only as a column type declaration in the table definition of DDL and cannot be directly called in queries. The return value is of the TIMESTAMP_LTZ(3) type, indicating the machine time when the data is processed.	`-- Using in DDL: proctime AS PROCTIME()`

6.6 Conditional Functions

Overview

Conditional functions return different values based on the condition expression. **CASE WHEN** supports multi-condition branch logic, **IF** implements simple conditional value selection, **COALESCE** is used to replace null values, **NULLIF** is used to set equivalent values to null, **ISNULL**/**ISNOTNULL** is used to determine null values, and **GREATEST**/**LEAST** is used to select extreme values from multiple values. AI DataLake supports all conditional functions.

For more information, see **Conditional Functions**.

 Copyright © Huawei Technologies Co., Ltd.

Application Scenarios

- **Data classification**: **CASE WHEN** is used to map continuous values to discrete categories. For example, amounts are classified into high, medium, and low tiers.
- **Conditional value selection**: The **IF** function is used to quickly select between two values. For example, different status texts are returned based on the flag.
- **Null value replacement**: **COALESCE** is used to replace **NULL** with a default value to prevent **NULL** from participating in calculations and causing the result to be **NULL**.
- **Equality setting to null**: **NULLIF** is used to return **NULL** when two values are equal. A typical use case is to avoid division by zero errors.
- **Null value judgment**: **ISNULL** or **ISNOTNULL** in **WHERE** conditions or **CASE WHEN** is used to determine whether a value is **NULL**.
- **Extreme value selection**: **GREATEST** or **LEAST** is used to select the maximum or minimum value from multiple columns or expressions, automatically skipping **NULL**.

Notes and Constraints

- The return types of all **WHEN**/**THEN** and **ELSE** must be compatible. The system automatically derives the common type. If the **ELSE** clause is missing and no condition is met, **NULL** is returned.
- The types of **true_value** and **false_value** must be compatible. The system automatically derives the common type. The **condition** must be of the BOOLEAN type.
- At least two parameters are required. All parameter types must be compatible, and the return type is the common type of all parameters.
- The types of **value1** and **value2** must be comparable. **NULL** is returned only when **value1 = value2** is **TRUE**. If the comparison result is **NULL**, **value1** is returned.
- At least two parameters are required. All parameter types must be compatible and sortable. **NULL** values are automatically skipped. If all values are **NULL**, **NULL** is returned.

 Copyright © Huawei Technologies Co., Ltd.

Syntax Structure

Syntax	Return Type	Description	Example
CASE WHEN condition1 THEN result1 [WHEN condition2 THEN result2]* [ELSE result] END	Consistent with **result**	Search condition. The conditions are evaluated in sequence, and the first result that meets the condition (condition is **TRUE**) is returned. If none of the conditions is met and the **ELSE** clause exists, the **ELSE** value is returned. If none of the conditions is met and the **ELSE** clause does not exist, **NULL** is returned. The condition must be of the BOOLEAN type. All result types must be compatible.	SELECT CASE WHEN 1 > 2 THEN 'a' WHEN 2 > 1 THEN 'b' ELSE 'c' END; -- 'b'
CASE value WHEN value1 THEN result1 [WHEN value2 THEN result2]* [ELSE result] END	Consistent with **result**	Value matching condition. The **value** is compared with each **valueN** following **WHEN** in sequence, and the first matching result is returned. If none of the values is matched and the **ELSE** clause exists, the **ELSE** value is returned. If none of the values is matched and the **ELSE** clause does not exist, **NULL** is returned. The **value** and each **valueN** must be of the comparable type. All **result** types must be compatible.	SELECT CASE 2 WHEN 1 THEN 'one' WHEN 2 THEN 'two' ELSE 'other' END; -- 'two'
IF(condition, true_value, false_value)	Consistent with the common type of **true_value** and **false_value**.	Conditional value function. Returns **true_value** if **condition** is **TRUE**, and returns **false_value** if condition is **FALSE**. The **condition** must be of the BOOLEAN type. The types of **true_value** and **false_value** must be compatible. If **condition** is **NULL**, **false_value** is returned.	SELECT IF(1 > 2, 'a', 'b'); -- 'b'

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
COALESCE(value1, value2, ...)	Consistent with the first non-null value	Returns the first non-null value in the parameter list. At least two parameters are required. All parameter types must be compatible. If all parameters are **NULL**, **NULL** is returned.	SELECT COALESCE(NULL, NULL, 'default'); -- 'default'
NULLIF(value1, value2)	Consistent with **value1** or **NULL**	Null-if function. Returns **NULL** if **value1** is equal to **value2**; otherwise, returns **value1**. The types of **value1** and **value2** must be comparable. This function is often used to avoid division-by-zero errors (for example, **amount / NULLIF(count, 0)**).	SELECT NULLIF(5, 5); -- NULL
ISNULL(value)	BOOLEAN	Determines whether a value is null. Returns **TRUE** if value is **NULL**, and returns **FALSE** otherwise. This function is equivalent to **value IS NULL** and is not affected by the rule that **NULL** is returned when **NULL** participates in a general comparison.	SELECT ISNULL(NULL); -- TRUE
ISNOTNULL(value)	BOOLEAN	Determines whether a value is not null. If the value is not **NULL**, **TRUE** is returned. Otherwise, **FALSE** is returned. This function is equivalent to value **IS NOT NULL** and is the logical negation of **ISNULL**.	SELECT ISNOTNULL(1); -- TRUE
GREATEST(value1, value2, ...)	Consistent with the common type of the parameters.	Returns the maximum value in the parameter list, automatically skipping **NULL**. At least two parameters are required. All parameter types must be compatible and sortable. If all values are **NULL**, **NULL** is returned.	SELECT GREATEST(3, 1, 4, 1, 5); -- 5

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
LEAST(value1, value2, ...)	Consistent with the common type of the parameters.	Returns the minimum value in the parameter list, automatically skipping **NULL**. At least two parameters are required. All parameter types must be compatible and sortable. If all values are **NULL**, **NULL** is returned.	SELECT LEAST(3, 1, 4, 1, 5); -- 1

6.7 Type Conversion Functions

Overview

Type conversion functions are used to convert between different data types in Flink SQL. These functions can convert a value of one data type to another, or obtain the data type information of an expression at runtime. Type conversion is a core operation in data processing, cleansing, and format adaptation, and is widely used in scenarios such as ETL pipelines, heterogeneous data source integration, and dynamic type determination.

For more information, see **Type Conversion Functions**.

Application Scenarios

- **ETL data cleansing**: After reading data from heterogeneous data sources, uses **CAST** or **TRY_CAST** to convert string fields to target types such as numeric and date types.
- **Data quality fault tolerance**: Uses **TRY_CAST** for fields that may contain dirty data to prevent the processing of the entire record from being interrupted due to individual conversion failures.
- **Dynamic type determination**: Uses **TYPEOF** to obtain the data type of a field or expression at runtime for conditional branching or debugging and troubleshooting.
- **Cross-source data alignment**: When field types from different data sources are inconsistent, uses type conversion functions to unify the data types before performing **JOIN** or aggregation operations.

Notes and Constraints

- The target type of **CAST** and **TRY_CAST** must be a data type keyword supported by Flink SQL, such as **INT**, **BIGINT**, **STRING**, **DOUBLE**, **DECIMAL**, and **TIMESTAMP**.
- Not all types can be converted to each other. For example, direct conversion between **ARRAY** and **MAP**, or between **ROW** and **INT**, is not supported.

 Copyright © Huawei Technologies Co., Ltd.

Converting an unsupported type pair throws an exception (**CAST**) or returns **NULL** (**TRY_CAST**).

- When converting from **STRING** to a numeric type, the string must conform to the format requirements of the target type. For example, **CAST('abc' AS INT)** throws an exception, while **TRY_CAST('abc' AS INT)** returns **NULL**.
- When converting from a numeric type to **STRING**, the result format is determined by the system by default, and the original precision information may not be retained.
- The type string returned by **TYPEOF** is the internal type representation of Flink. Its format may vary with the version, and you should not hardcode dependencies on the specific content of this string in your business logic.
- In **TYPEOF(input, forceSerializable)**, **forceSerializable** is of the BOOLEAN type. When set to **TRUE**, the returned type string is guaranteed to be serializable, but some details of non-serializable types may be omitted.

Syntax Structure of Type Conversion

Syntax	Return Type	Description	Example
CAST(value AS type)	Determined by **type**	Converts the value to the data type specified by **type**. The value is the expression to be converted, and type is the keyword of the target data type (such as **INT**, **BIGINT**, **STRING**, **DOUBLE**, **DECIMAL(p, s)**, **BOOLEAN**, **DATE**, and **TIMESTAMP**). If the conversion fails, a runtime exception is thrown. Valid conversions between all Flink SQL data types are supported. An error is reported for unsupported type pairs.	`SELECT CAST('123' AS INT); -- 123`
TRY_CAST(value AS type)	Determined by **type**. If the conversion fails, **NULL** is returned.	Converts the value to the data type specified by **type**. The parameter description is the same as that of **CAST**. The difference between this function and **CAST** is that when the conversion fails (for example, the type is incompatible or the value format does not match), no exception is thrown, but **NULL** is returned. This function is applicable to scenarios where the data quality is uncertain and fault tolerance is required.	`SELECT TRY_CAST('abc' AS INT); -- NULL`

 Copyright © Huawei Technologies Co., Ltd.

Syntax Structure of Type Query

Syntax	Return Type	Description	Example
TYPEOF(input)	STRING	Returns the string representation of the data type of the input expression. The **input** is an expression of any type. The return value reflects the information about the expression type in Flink, in the format of **INTEGER**, **VARCHAR(2147483647)**, or **ROW<f1 INT, f2 STRING>**.	SELECT TYPEOF(123); -- INTEGER
TYPEOF(input, forceSerializable)	STRING	Returns the string representation of the data type of the input expression. The **input** is an expression of any type, and **forceSerializable** is of the BOOLEAN type. When **forceSerializable** is set to **TRUE**, the returned type string is guaranteed to be serializable, which is applicable to scenarios where type information needs to be transmitted over the network or persisted. When **forceSerializable** is set to **FALSE**, the behavior is the same as that of the single-parameter version.	SELECT TYPEOF(123, TRUE); -- INTEGER

6.8 Aggregate Functions

Overview

Aggregate functions calculate a group of input values and return a single result. Aggregate functions are usually used together with the **GROUP BY** clause to aggregate data by group key and calculate the result of each group. They can also be used as window aggregate functions in **GROUP WINDOW** or **OVER WINDOW**. All aggregate functions (except **COUNT(*)**) ignore **NULL** values by default during calculation. You can use the **DISTINCT** keyword to remove duplicate values before aggregation.

For more information, see **Aggregate Functions**.

 Copyright © Huawei Technologies Co., Ltd.

Application Scenarios

- Collects statistics on metrics by dimension, for example, collecting statistics on the number of orders and average amount by region.
- Performs window aggregate calculation to calculate tumbling or sliding metrics, for example, calculating the UV and GMV in the last 5 minutes.
- Data deduplication statistics, for example, **COUNT(DISTINCT user_id)** calculates the number of unique visitors.
- Aggregates multi-value fields, for example, collecting labels in a group into a set or concatenating multiple rows of text into one row.
- Identifies the aggregation level in **ROLLUP**/**CUBE** multidimensional analysis.

Notes and Constraints

- **NULL value processing**: Except **COUNT(*)**, all aggregate functions automatically skip **NULL** values during calculation. If all values in a group are **NULL**, the aggregation result is **NULL**.
- **DISTINCT restriction**: **DISTINCT** supports only the **COUNT**, **AVG**, **SUM**, and **COLLECT** functions. **MAX**, **MIN**, **STDDEV_POP**, **STDDEV_SAMP**, **VAR_POP**, **VAR_SAMP**, **LISTAGG**, **GROUPING**, **BIT_AND**, and **BIT_OR** do not support **DISTINCT**.
- **Nested aggregation**: Aggregate functions cannot be nested. For example, **SUM(COUNT(*))** is invalid and must be implemented through subqueries.
- **GROUP BY consistency**: When a non-aggregated column is referenced in the **SELECT** clause, the column must appear in the **GROUP BY** clause.
- **Windowed aggregation**: When an aggregate function is used in an **OVER WINDOW** clause, it must be used together with **ORDER BY** and a window frame (**ROWS BETWEEN**/**RANGE BETWEEN**).
- **Minimum row count for STDDEV_SAMP/VAR_SAMP**: The sample standard deviation and sample variance require at least two non-null values in a group. Otherwise, **NULL** is returned.
- **Type constraints for BIT_AND/BIT_OR**: Only integer types are supported as input. Passing a non-integer type causes type validation to fail.
- **Return type of COLLECT**: **COLLECT** returns the MULTISET type. To convert it to an array, you can use **ARRAY_DISTINCT(COLLECT(...))** or other methods.
- **Separator of LISTAGG**: The **separator** parameter must be a string constant. Column references or expressions are not supported.

Syntax Structure

Syntax	Return Type	Description	Example
COUNT(*)	BIGINT	Counts the total number of rows, **including rows with NULL values**. You do not need to specify columns. This function counts the entire row.	SELECT COUNT(*) FROM orders; -- 10

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
COUNT(value)	BIGINT	Counts the number of rows in which the **value** is not **NULL**. The parameter can be of any type. Rows with **NULL** values are not counted.	SELECT COUNT(amount) FROM orders; -- 8 (skipping 2 rows with NULL values)
COUNT(DISTINCT value)	BIGINT	Counts the number of distinct values after **NULL** values are removed. The parameter can be of any type.	SELECT COUNT(DISTINCT user_id) FROM orders; -- 5
AVG(value)	DECIMAL / DOUBLE	Calculates the arithmetic average of **value**. The parameter must be of a numeric type (TINYINT/ SMALLINT/INT/ BIGINT/FLOAT/ DOUBLE/DECIMAL). For integer input, the return type is DECIMAL. For floating-point input, the return type is DOUBLE. **NULL** values are skipped.	SELECT AVG(price) FROM orders; -- 25.50
AVG(DISTINCT value)	DECIMAL / DOUBLE	Calculates the arithmetic average of **value** after removing **NULL** and duplicate values. The parameter must be of a numeric type. The return type rules are the same as those for AVG.	SELECT AVG(DISTINCT price) FROM orders; -- 30.00

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
SUM(value)	Consistent with the input type	Calculates the sum of **value**. The parameter must be of a numeric type. For integer input, the return type is integer (INT → INT, BIGINT → BIGINT). For DECIMAL input, the return type is DECIMAL. For FLOAT/DOUBLE input, the return type is DOUBLE. **NULL** values are skipped.	SELECT SUM(amount) FROM orders; -- 1500
SUM(DISTINCT value)	Consistent with the input type	Calculates the sum of **value** after removing **NULL** and duplicate values. The parameter must be of a numeric type. The return type rules are the same as those for SUM.	SELECT SUM(DISTINCT amount) FROM orders; -- 1200
MAX(value)	Consistent with the input type	Returns the maximum value of **value**. The parameter can be of any comparable type (such as number, string, or date and time). **NULL** values are skipped. Strings are compared in lexicographic order.	SELECT MAX(price) FROM orders; -- 99.90
MIN(value)	Consistent with the input type	Returns the minimum value of **value**. The parameter can be of any comparable type (such as number, string, or date and time). **NULL** values are skipped. Strings are compared in lexicographic order.	SELECT MIN(price) FROM orders; -- 5.00

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
STDDEV_POP(value)	DOUBLE	Calculates the **population standard deviation** of **value**. The parameter must be of the numeric type. The formula is **$\sqrt{\Sigma(x-\mu)^2/N}$**. **NULL** values are skipped. If the number of input rows is less than 1, **NULL** is returned.	SELECT STDDEV_POP(score) FROM exam; -- 12.45
STDDEV_SAMP(value)	DOUBLE	Calculates the **sample standard deviation** of **value**. The parameter must be of the numeric type. The formula is **$\sqrt{\Sigma(x-\mu)^2/(N-1)}$**. **NULL** values are skipped. If the number of input rows is less than 2, **NULL** is returned.	SELECT STDDEV_SAMP(score) FROM exam; -- 13.20
VAR_POP(value)	DOUBLE	Calculates the **population variance** of **value**. The parameter must be of the numeric type. The formula is **$\Sigma(x-\mu)^2/N$**. **NULL** values are skipped. If the number of input rows is less than 1, **NULL** is returned.	SELECT VAR_POP(score) FROM exam; -- 155.00
VAR_SAMP(value)	DOUBLE	Calculates the **sample variance** of **value**. The parameter must be of the numeric type. The formula is **$\Sigma(x-\mu)^2/(N-1)$**. **NULL** values are skipped. If the number of input rows is less than 2, **NULL** is returned.	SELECT VAR_SAMP(score) FROM exam; -- 174.38

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
COLLECT(value)	MULTISET	Collects values into a multiset, and retains duplicate elements. The parameter can be of any type. **NULL** values are skipped. The return type is MULTISET<parameter type>.	SELECT COLLECT(tag) FROM articles; -- [a, b, b, c]
COLLECT(DISTINCT value)	MULTISET	Collects values into a multiset after removing **NULL** values and duplicates. The parameter can be of any type. The return type is MULTISET<parameter type>.	SELECT COLLECT(DISTINCT tag) FROM articles; -- [a, b, c]
LISTAGG(value, separator)	STRING	Concatenates **value** into a single string using **separator**. The **value** must be of the string type, and the **separator** must be a string constant. **NULL** values are skipped. If there is no separator, pass an empty string '' as the **separator**.	SELECT LISTAGG(name, ',') FROM users; -- Alice,Bob,Carol

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
GROUPING(expression)	INT	Determines whether the **expression** is populated with **NULL** values due to aggregation by **GROUP BY ROLLUP** or **GROUP BY CUBE**. A return value of **1** indicates that the column is aggregated (the value is produced by **ROLLUP**/**CUBE**), and a return value of **0** indicates that the column retains the original grouping value. This function must be used in a **GROUP BY** clause that includes **ROLLUP** or **CUBE**.	SELECT year, region, GROUPING(region) AS g FROM sales GROUP BY ROLLUP (year, region); -- g=0 (original row) or g=1 (subtotal row)
BIT_AND(value)	INT / BIGINT	Performs bitwise **AND** aggregation on **value**. The parameter must be of an integer type (**TINYINT**/**SMALLINT**/**INT**/**BIGINT**). This function is equivalent to performing the & operation on all non-NULL values sequentially. **NULL** values are skipped. If there is only one non-NULL value, that value itself is returned.	SELECT BIT_AND(flags) FROM events; -- 8

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
BIT_OR(value)	INT / BIGINT	Performs bitwise **OR** aggregation on **value**. The parameter must be of the integer type (**TINYINT**/**SMALLINT**/**INT**/**BIGINT**). This function is equivalent to performing the OR operation on all non-NULL values in sequence. **NULL** values are skipped. If there is only one non-NULL value, the value itself is returned.	`SELECT BIT_OR(flags) FROM events;` `-- 15`

6.9 JSON Functions

Overview

JSON functions are used to parse, query, construct, and validate JSON strings. AI DataLake supports JSON path expressions defined in the SQL standard (ISO/IEC TR 19075-6), and supports two modes: lax and strict. It provides complete JSON processing capabilities, including JSON validity determination, path query, scalar extraction, and JSON object and array construction.

For more information, see **JSON Functions**.

Application Scenarios

- **JSON data validation**: Uses **IS JSON** to determine whether a string is a valid JSON string, or whether it is a specific JSON type.
- **JSON path query**: Uses **JSON_EXISTS** to determine whether a path exists, uses **JSON_VALUE** to extract scalar values, and uses **JSON_QUERY** to extract JSON fragments.
- **JSON serialization**: Uses **JSON_STRING** to serialize values into JSON strings.
- **JSON construction**: Uses **JSON_OBJECT** and **JSON_ARRAY** to dynamically construct JSON objects and arrays.

Notes and Constraints

- The JSON path expression uses symbols to represent the root node. It supports attribute access (**.***a*), array element access (**.***a***[0].***b*), and array traversal (**.***a***[*].***b*).
- There are two path modes: lax and strict. In lax mode, errors are more tolerant and are converted to empty sequences. In strict mode, data is checked from the schema perspective, and an error is thrown when data that

 Copyright © Huawei Technologies Co., Ltd.

does not comply with the path expression is encountered. If this parameter is omitted, the strict mode is used by default.

- **JSON_VALUE** can extract only scalar values. Non-scalar values cannot be returned. The following return types are supported: **VARCHAR**/**STRING**, **BOOLEAN**, **INTEGER**, and **DOUBLE**.
- The key of **JSON_OBJECT** must be a non-NULL string literal, and the value can be any expression.
- If **ON NULL** is omitted in **JSON_ARRAY**, **ABSENT ON NULL** (ignore **NULL** values) is used by default. If **ON NULL** is omitted in **JSON_OBJECT**, **NULL ON NULL** (include **NULL** as a NULL value) is used by default.
- Some functions in lax mode are not fully supported currently (upstream defect **CALCITE-4717**). Non-standard behaviors are not guaranteed.

Syntax Structure

Syntax	Return Type	Description	Example
IS JSON [{ VALUE | SCALAR | ARRAY | OBJECT }]	BOOLEAN	Determines whether a given string is valid JSON. The optional **type** parameter can be specified to restrict the allowed JSON object type. If the string is valid JSON but not of the specified type, **false** is returned. The default value is **VALUE**.	-- TRUE '1' IS JSON '[]' IS JSON '{}' IS JSON -- TRUE '"abc"' IS JSON -- FALSE 'abc' IS JSON NULL IS JSON -- TRUE '1' IS JSON SCALAR -- FALSE '1' IS JSON ARRAY -- FALSE '1' IS JSON OBJECT -- FALSE '{}' IS JSON SCALAR -- FALSE '{}' IS JSON ARRAY -- TRUE '{}' IS JSON OBJECT
JSON_EXISTS(jsonValue, path [{ TRUE | FALSE | UNKNOWN | ERROR } ON ERROR])	BOOLEAN	Determines whether a JSON string satisfies a given path search condition. If the error handling behavior is omitted, the default value is **FALSE ON ERROR**.	-- TRUE SELECT JSON_EXISTS('{"a": true}', '$.a'); -- FALSE SELECT JSON_EXISTS('{"a": true}', '$.b'); -- TRUE SELECT JSON_EXISTS('{"a": [{ "b": 1 }]}', '$.a[0].b'); -- TRUE SELECT JSON_EXISTS('{"a": true}', 'strict $.b' TRUE ON ERROR); -- FALSE SELECT JSON_EXISTS('{"a": true}', 'strict $.b' FALSE ON ERROR);

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
JSON_STRING(value)	STRING	Serializes a value to JSON. This function returns a JSON string containing the serialized value. If the value is **NULL**, the function returns **NULL**.	-- NULL JSON_STRING(CAST(NULL AS INT)) -- '1' JSON_STRING(1) -- 'true' JSON_STRING(TRUE) -- '"Hello, World!"' JSON_STRING('Hello, World!') -- '[1,2]' JSON_STRING(ARRAY[1, 2])

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
JSON_VALUE(jsonValue, path [RETURNING] [{ NULL | ERROR | DEFAULT } ON EMPTY] [{ NULL | ERROR | DEFAULT } ON ERROR])	STRING / BOOLEAN / INTEGER / DOUBLE	Extracts scalar values from JSON strings. This method searches for the given path expression in the JSON string and returns the value if the value at the path is a scalar. Non-scalar values cannot be returned. By default, the value is returned as a STRING. You can use **dataType** to select a different type. The supported types are **VARCHAR**/**STRING**, **BOOLEAN**, **INTEGER**, and **DOUBLE**. For empty path expressions or errors, you can define the behavior to return NULL, throw an error, or return a defined default value. If omitted, the default behavior is **NULL ON EMPTY** or **NULL ON ERROR**, respectively. The default value can be a literal or an expression. If the default value itself throws an error, the error behavior of **ON EMPTY** is used, and an error is thrown for **ON ERROR**. If the path contains special characters such as spaces, you can use **['property']** or **["property"]** to select the specified property in the parent object. You must enclose the property name with single or double quotation marks. When **JSON_VALUE** is used in SQL, the path is a single quoted character parameter. Therefore, you need to escape the single quotation marks around the property name, for example,	-- "true" JSON_VALUE('{"a": true}', '$.a') -- TRUE JSON_VALUE('{"a": true}', '$.a' RETURNING BOOLEAN) -- "false" JSON_VALUE('{"a": true}', 'lax $.b' DEFAULT FALSE ON EMPTY) -- "false" JSON_VALUE('{"a": true}', 'strict $.b' DEFAULT FALSE ON ERROR) -- 0.998D JSON_VALUE('{"a.b": [0.998,0.996]}','$.["a.b"][0]' RETURNING DOUBLE) -- "right" JSON_VALUE('{"contains blank": "right"}', 'strict $.[''contains blank'']' NULL ON EMPTY DEFAULT 'wrong' ON ERROR)

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
		JSON_VALUE('{"a b": "true"}', '$.["a b"]').	
JSON_QUERY(jsonValue, path [RETURNING] [{ WITHOUT | WITH CONDITIONAL | WITH UNCONDITIONAL } [ARRAY] WRAPPER] [{ NULL | EMPTY ARRAY | EMPTY OBJECT | ERROR } ON EMPTY] [{ NULL | EMPTY ARRAY | EMPTY OBJECT | ERROR } ON ERROR])	STRING / ARRAY<STRING>	Extracts JSON values from a JSON string. The result is returned as a STRING or ARRAY<STRING> type. You can control the result using the **RETURNING** clause. The **wrappingBehavior** determines whether the extracted value should be wrapped as an array, and whether it should be wrapped unconditionally or only when the value itself is not an array. **onEmpty** and **onError** determine the behavior when the path expression is empty or an error occurs. By default, **NULL** is returned in both cases. Other options include using an empty array, an empty object, or throwing an error.	-- '{ "b": 1 }' JSON_QUERY('{ "a": { "b": 1 } }', '$.a') -- '[1, 2]' JSON_QUERY('[1, 2]', '$') -- NULL JSON_QUERY(CAST(NULL AS STRING), '$') -- '["c1","c2"]' JSON_QUERY('{"a":[{"c":"c1"}, {"c":"c2"}]}', 'lax $.a[*].c') -- Wrap result into an array. -- '[{}]' JSON_QUERY('{}', '$' WITH CONDITIONAL ARRAY WRAPPER) -- '[1, 2]' JSON_QUERY('[1, 2]', '$' WITH CONDITIONAL ARRAY WRAPPER) -- '[[1, 2]]' JSON_QUERY('[1, 2]', '$' WITH UNCONDITIONAL ARRAY WRAPPER) -- Scalars must be wrapped to return. -- NULL JSON_QUERY(1, '$') -- '[1]' JSON_QUERY(1, '$' WITH CONDITIONAL ARRAY WRAPPER) -- Behavior if path expression is empty/there is an error. -- '{}' JSON_QUERY('{}', 'lax $.invalid' EMPTY OBJECT ON EMPTY) -- '[]' JSON_QUERY('{}', 'strict $.invalid' EMPTY ARRAY ON ERROR)

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
JSON_OBJECT([[KEY] key VALUE value]* [{ NULL | ABSENT } ON NULL])	STRING	Builds a JSON object string from a list of key-value pairs. Note that the **key** must be a non-NULL string literal, while the **value** can be any expression. This function returns a JSON string. The **ON NULL** behavior defines how **NULL** values are handled. If omitted, the default is **NULL ON NULL**. Values created from other JSON constructors (**JSON_OBJECT**, **JSON_ARRAY**) are inserted directly, not as strings. This allows for the construction of nested JSON structures.	-- '{}' JSON_OBJECT() -- '{"K1":"V1","K2":"V2"}' JSON_OBJECT('K1' VALUE 'V1', 'K2' VALUE 'V2') -- Expressions as values JSON_OBJECT('orderNo' VALUE orders.orderId) -- ON NULL JSON_OBJECT(KEY 'K1' VALUE CAST(NULL AS STRING) NULL ON NULL) -- '{"K1":null}' JSON_OBJECT(KEY 'K1' VALUE CAST(NULL AS STRING) ABSENT ON NULL) -- '{}' -- '{"K1":{"K2":"V"}}' JSON_OBJECT(KEY 'K1' VALUE JSON_OBJECT(KEY 'K2' VALUE 'V'))
JSON_ARRAY([value]* [{ NULL | ABSENT } ON NULL])	STRING	Builds a JSON array string from a list of values. This function returns a JSON string. The **value** can be any expression. The **ON NULL** behavior defines how **NULL** values are handled. If omitted, the default is **ABSENT ON NULL**. Elements created from other JSON constructors (**JSON_OBJECT**, **JSON_ARRAY**) are inserted directly, not as strings. This allows for the construction of nested JSON structures.	-- '[]' JSON_ARRAY() -- '[1,"2"]' JSON_ARRAY(1, '2') -- Expressions as values JSON_ARRAY(orders.orderId) -- ON NULL JSON_ARRAY(CAST(NULL AS STRING) NULL ON NULL) -- '[null]' JSON_ARRAY(CAST(NULL AS STRING) ABSENT ON NULL) -- '[]' -- '[[1]]' JSON_ARRAY(JSON_ARRAY(1))

6.10 Value Construction Functions

Overview

Value construction functions are used to construct values of composite data types, including rows, arrays, and maps. AI DataLake supports the implicit and explicit

 Copyright © Huawei Technologies Co., Ltd.

row construction, array construction, and map construction capabilities in the SQL standard.

For more information, see **Value Construction Functions**.

Application Scenarios

- **Composite row construction**: The **ROW** construction function is used to create row values that contain multiple columns.
- **Array construction**: The **ARRAY** construction function is used to create array values, which are used in the **IN** clause and **UNNEST** operation.
- **Mapping construction**: The **MAP** construction function is used to create key-value pair maps, which are used to configure parameters and store attributes.
- **Window definition**: The row number interval construction function is used to assist window definition.

Notes and Constraints

- Implicit row construction (using parentheses) requires at least two columns. The explicit ROW construction function can accept any number of columns.
- The value types in the **ARRAY** and **MAP** construction functions must be consistent or compatible.
- The parameters of the **MAP** construction function must appear in pairs. That is, the number of key-value pairs must be an even number.
- The row interval construction function (*N* rows) is supported only by Table API and is not available in SQL.

Syntax Structure

Syntax	Return Type	Description	Example
implicit constructor with parenthesis (value1 [, value2]*) **explicit** ROW constructor with ROW(value1 [, value2]*)	ROW	Creates a row from a list of values (**value1**, **value2**, ...). The implicit row constructor requires at least two fields. The explicit row constructor can handle any number of fields. Both support any expression as a field.	-- Implicit row constructor (at least two fields) SELECT (1, 'hello'); -- Explicit row constructor (can handle any number of fields) SELECT ROW(1); SELECT ROW(1, 'hello', TRUE); -- Using expressions as fields SELECT ROW(orderId, orderAmount) FROM orders;

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
ARRAY '[' value1 [, value2]* ']'	ARRAY	Creates an array from a list of values (**value1**, **value2**, ...).	-- Create an array from literals. SELECT ARRAY [1, 2, 3]; -- Create an array from expressions. SELECT ARRAY [orderId, orderAmount] FROM orders; -- Create a nested array. SELECT ARRAY [ARRAY [1, 2], ARRAY [3, 4]];
MAP '[' value1, value2 [, value3, value4]* ']'	MAP	Creates a map from a list of key-value pairs ((**value1**, **value2**), (**value3**, **value4**), ...).	-- Create a map from literals. SELECT MAP ['a', 1, 'b', 2]; -- Create a map from expressions. SELECT MAP ['orderId', orderId, 'amount', orderAmount] FROM orders;

6.11 Value Access Functions

Overview

Value access functions are used to access field values in composite data types (such as Tuple and POJO). They support accessing a single field by name and expanding all subfields. AI DataLake supports the capability of accessing composite type fields using the dot notation in the SQL standard.

For more information, see **Value Access Functions**.

Application Scenarios

- **Field access**: Obtains the value of a specific field from a composite type (such as Tuple and POJO) by name.
- **Field expansion**: Expands all direct subtypes of a composite type into independent fields, used to expand nested structures in **SELECT** queries.

Notes and Constraints

- The dot access syntax (**tableName.compositeType.field**) is only applicable to Flink composite types (such as Tuple and POJO) and is not applicable to common scalar fields.
- The expand operation (**tableName.compositeType.***) converts each direct subtype into an independent field. The field name is similar to the original field name, but separated by dollar signs (for example, **mypojo$mytuple$f0**).
- The expand operation only expands direct subtypes and does not recursively expand nested composite types.

 Copyright © Huawei Technologies Co., Ltd.

Syntax Structure

Syntax	Return Type	Description	Example
tableName.compositeType.field	ANY	Returns the value of a field in a Flink composite type (such as Tuple and POJO) by name.	-- Assume that the table **myTable** contains the ROW type field **myRow(field1 INT, field2 STRING)**. -- Access the **field1** field by name. SELECT myTable.myRow.field1 FROM myTable; -- Access the field in the nested structure by name. -- Assume that **person** is **ROW(name STRING, age INT)**. SELECT myTable.person.name FROM myTable;
tableName.compositeType.*	ANY	Returns a flattened representation of a Flink composite type (such as Tuple and POJO), converting each direct subtype into an independent field. In most cases, the flattened field name is similar to the original field name, but uses the dollar sign ($) as the separator (for example, **mypojo $mytuple$f0**).	-- Assume that the table **myTable** contains the ROW type field **myRow(field1 INT, field2 STRING)**. -- All fields in **myRow** are flattened. The flattened field names are separated by dollar signs ($) (for example, **mypojo $mytuple$f0**). SELECT myTable.myRow.* FROM myTable;

6.12 Grouping Functions

Overview

Grouping functions are used to identify grouping levels in **GROUP BY** extensions (such as **ROLLUP**, **CUBE**, and **GROUPING SETS**). AI DataLake supports the **GROUP_ID** and **GROUPING**/**GROUPING_ID** functions in the SQL standard, which are used to distinguish different group combinations and determine whether a column is involved in the current group aggregation.

For more information, see **Grouping Functions**.

Application Scenarios

- **Multi-level aggregation reports**: Identifies different group combinations in multi-level group aggregation, such as **ROLLUP**, **CUBE**, and **GROUPING SETS**.
- **Distinguishing subtotals from totals**: Uses the **GROUPING** function to determine whether a column is aggregated in the current group (**NULL** is returned) to distinguish original data rows from aggregated rows.

 Copyright © Huawei Technologies Co., Ltd.

- **Condition filtering**: Uses the **GROUPING** result in the **HAVING** clause to filter specific group levels.

Notes and Constraints

- The **GROUP_ID** and **GROUPING**/**GROUPING_ID** functions are only meaningful in **GROUP BY** statements that contain **ROLLUP**, **CUBE**, or **GROUPING SETS**.
- **GROUPING_ID** is an alias of **GROUPING**, and they are semantically equivalent.
- In the bit vector returned by **GROUPING**, **0** indicates that the corresponding column is involved in **GROUP BY**, and **1** indicates that the corresponding column is not involved in **GROUP BY** (the column is aggregated to **NULL**).

Syntax Structure

Syntax	Return Type	Description	Example
GROUP_ID()	INTEGER	Returns an integer that uniquely identifies the combination of grouping keys.	-- Used together with **ROLLUP** to identify each grouping level. SELECT dept, role, GROUP_ID() AS gid, COUNT(*) AS cnt FROM employees GROUP BY ROLLUP(dept, role); -- Example result: -- dept=Sales, role=Manager, gid=0, cnt=5 -- dept=Sales, role=NULL, gid=0, cnt=10 -- dept=NULL, role=NULL, gid=0, cnt=50
GROUPING(expression1 [, expression2]*) GROUPING_ID(expression1 [, expression2]*)	INTEGER	Returns a bit vector for the given grouping expression.	-- Used together with **ROLLUP** to determine whether the grouping key is aggregated. SELECT dept, role, GROUPING(dept, role) AS grp_bits, COUNT(*) AS cnt FROM employees GROUP BY ROLLUP(dept, role); -- Example result: -- dept=Sales, role=Manager, grp_bits=0, cnt=5 -- dept=Sales, role=NULL, grp_bits=1, cnt=10 -- dept=NULL, role=NULL, grp_bits=3, cnt=50 -- Uses **CASE WHEN** to distinguish aggregation levels. SELECT CASE WHEN GROUPING (dept, role) = 0 THEN 'Details' WHEN GROUPING(dept, role) = 1 THEN 'Department subtotal' WHEN GROUPING (dept, role) = 3 THEN 'Total' END AS level, dept, role, COUNT(*) AS cnt FROM employees GROUP BY ROLLUP(dept, role);

6.13 Hash Functions

Overview

Hash functions are used to calculate hash values of strings. Common hash algorithms such as MD5, SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512 are

 Copyright © Huawei Technologies Co., Ltd.

supported. AI DataLake supports the hash calculation capability in the SQL standard and returns hash values in hexadecimal strings.

For more information, see **Hash Functions**.

Application Scenarios

- **Data verification**: Uses MD5 or SHA hashes to verify data integrity.
- **Sensitive information anonymization**: Performs hash anonymization on sensitive fields (such as ID card numbers and mobile numbers).
- **Data partitioning**: Uses hash values for data sharding or distributed partition key generation.
- **Fingerprint comparison**: Uses hash values to quickly determine whether data is consistent.

Notes and Constraints

- The input parameters of all hash functions must be of the STRING type. If the input is **NULL**, **NULL** is returned.
- MD5 returns a 32-digit hexadecimal string, SHA-1 returns a 40-digit string, SHA-224 returns a 56-digit string, SHA-256 returns a 64-digit string, SHA-384 returns a 96-digit string, and SHA-512 returns a 128-digit string.
- The **hashLength** parameter of the SHA2 function supports only the values **224**, **256**, **384**, and **512**. An error is reported if other values are used.
- Hash functions are one-way functions and are irreversible. They are not suitable for encryption scenarios.

Syntax Structure

Syntax	Return Type	Description	Example
MD5(string)	STRING	Returns the MD5 hash value of a string, which is a 32-digit hexadecimal string. If the string is **NULL**, **NULL** is returned.	-- Returns '5d41402abc4b2a76b9719d911017c592'. SELECT MD5('hello'); -- Calculates the hash value for a table field. SELECT userId, MD5(userEmail) AS emailHash FROM users; -- Returns **NULL** if the input is **NULL**. SELECT MD5(CAST(NULL AS STRING));
SHA1(string)	STRING	Returns the SHA-1 hash value of a string, which is a 40-digit hexadecimal string. If the string is **NULL**, **NULL** is returned.	-- Returns 'aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d'. SELECT SHA1('hello'); -- Calculates the hash value for a table field. SELECT orderId, SHA1(orderDetail) AS detailHash FROM orders;

 Copyright © Huawei Technologies Co., Ltd.

Syntax	Return Type	Description	Example
SHA224(string)	STRING	Returns the SHA-224 hash value of a string, which is a 56-digit hexadecimal string. If the string is **NULL**, **NULL** is returned.	-- Returns a 56-digit hexadecimal string. SELECT SHA224('hello'); -- Calculates the hash value for a table field. SELECT userId, SHA224(userName) AS nameHash FROM users;
SHA256(string)	STRING	Returns the SHA-256 hash value of a string, which is a 64-digit hexadecimal string. If the string is **NULL**, **NULL** is returned.	-- Returns '2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824'. SELECT SHA256('hello'); -- Calculates the hash value for a table field. SELECT userId, SHA256(password) AS passwordHash FROM users; -- Returns **NULL** if the input is **NULL**. SELECT SHA256(CAST(NULL AS STRING));
SHA384(string)	STRING	Returns the SHA-384 hash value of a string, which is a 96-digit hexadecimal string. If the string is **NULL**, **NULL** is returned.	-- Returns a 96-digit hexadecimal string. SELECT SHA384('hello'); -- Calculates the hash value for a table field. SELECT docId, SHA384(docContent) AS contentHash FROM documents;
SHA512(string)	STRING	Returns the SHA-512 hash value of a string, which is a 128-digit hexadecimal string. If the string is **NULL**, **NULL** is returned.	-- Returns a 128-digit hexadecimal string. SELECT SHA512('hello'); -- Calculates the hash value for a table field. SELECT fileId, SHA512(fileContent) AS contentHash FROM files;
SHA2(string, hashLength)	STRING	Returns a hash value using the SHA-2 series hash functions (SHA-224, SHA-256, SHA-384, or SHA-512). The first parameter, **string**, is the string to be hashed, and the second parameter, **hashLength**, is the bit length of the result (**224**, **256**, **384**, or **512**). If **string** or **hashLength** is **NULL**, **NULL** is returned.	-- Uses SHA-256 (equivalent to the SHA256 function). Returns '2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824'. SELECT SHA2('hello', 256); -- Uses SHA-512 (equivalent to the SHA512 function). SELECT SHA2('hello', 512); -- Uses SHA-224 (equivalent to the SHA224 function). SELECT SHA2('hello', 224); -- Returns **NULL** when **hashLength** is **NULL**. SELECT SHA2('hello', CAST(NULL AS INT));

 Copyright © Huawei Technologies Co., Ltd.

7 Auxiliary Statements

7.1 USE Statement

Overview

The USE statement is used to switch the default catalog of the current session. After the switching, you can directly use the table name in subsequent SQL statements to reference the table, without the need to use the catalog name prefix.

Application Scenarios

- **Operations on multiple catalogs**: Switches between the Hive catalog and the default catalog to access different metadata sources.
- **Multi-tenant scenarios**: Different databases isolate data of different services. The **USE** statement can be used to quickly switch the working context.

Notes and Constraints

- The target catalog of the **USE CATALOG** statement must have been registered using the **CREATE CATALOG** statement.
- The **USE** statement affects only the current session and does not affect other sessions or global configurations.

Syntax Structure

-- Switch the current catalog.

```
USE CATALOG catalog_name;
```

Usage Example

Example: Switching the catalog using USE CATALOG

```
CREATE TABLE test_kafka (
  user_id varchar,
  item_id varchar,
  cat_id varchar,
```

 Copyright © Huawei Technologies Co., Ltd.

```
  test_time timestamp
) WITH (
  'properties.bootstrap.servers' = '<IP-address-of-Kafka-broker-instance>:<Kafka-port>',
  'format' = 'json',
  'topic' = 'zw_test_kafka',
  'connector' = 'kafka',
  'scan.startup.mode' = 'latest-offset'
);
CREATE CATALOG myhive WITH (
  'type' = 'hive',
  'hive-version' = '3.1.0',
  'default-database' = 'default',
  'hive-conf-dir' = 'xxxxxxx'
);
use catalog myhive;
set table.sql-dialect=hive;
CREATE TABLE IF NOT EXISTS user_behavior_hive_tbl (
    user_id STRING,
    item_id STRING,
    cat_id STRING,
    ts timestamp
  ) PARTITIONED BY (dy STRING, ho STRING, mi STRING) stored as textfile TBLPROPERTIES (
    'partition.time-extractor.timestamp-pattern' = '$dy $ho:$mi:00',
    'sink.partition-commit.trigger' = 'process-time',
    'sink.partition-commit.delay' = '0S',
    'sink.partition-commit.policy.kind' = 'metastore,success-file'
  );
set table.sql-dialect=default;
use catalog default_catalog;
INSERT into
  myhive.`default`.user_behavior_hive_tbl
SELECT
  user_id,
  item_id,
  cat_id,
  test_time,
  DATE_FORMAT(test_time, 'yyyy-MM-dd'),
  DATE_FORMAT(test_time, 'HH'),
  DATE_FORMAT(test_time, 'mm')
FROM
  test_kafka;
```

7.2 SET Statement

Overview

The **SET** statement is a session configuration management statement for Flink SQL. It is used to set the value of a specified configuration item. In AI DataLake, you can dynamically adjust job running parameters using the **SET** statement, affecting the execution behavior of subsequent SQL statements without restarting the job.

Application Scenarios

- **Adjusting the parallelism degree**: Modifies **parallelism.default** using the **SET** statement to control the parallelism degree of a job.
- **Switching the running mode**: Modifies **execution.runtime-mode** using the **SET** statement to switch between stream mode and batch mode.
- **Optimizing the state TTL**: Modifies **table.exec.state.ttl** using the **SET** statement to control the state retention duration, balancing performance and correctness.

 Copyright © Huawei Technologies Co., Ltd.

- **Configuring checkpoints**: Modifies the **execution.checkpointing.*** parameters using the **SET** statement to control checkpoint behavior.
- **Troubleshooting**: Views all current configurations using the **SET** statement to verify whether the parameters have taken effect.

Notes and Constraints

- The **SET** statement affects only the current session and is not persisted to the global configuration.
- Modifications to some configuration items take effect only for jobs submitted after the modification. Running jobs are not affected.
- The value set by the **SET** statement automatically becomes invalid after the session ends.
- The support scope of some configuration items on the AI DataLake platform may differ from that of open-source Flink. The actual environment prevails.

Syntax Structure

Modify configurations.

```
SET ('key' = 'value')
```

Common Configuration Parameters

Parameter	Default Value	Description
parallelism.default	1	Default job parallelism.
execution.runtime-mode	STREAMING	Running mode. The value can be **STREAMING** or **BATCH**.
table.exec.state.ttl	0 ms	State time-to-live (TTL). States that are not accessed within this period will be cleaned up.
table.exec.source.idle-timeout	0 ms	Source idle timeout interval. The idle marker is used for watermark advancement.
execution.checkpointing.interval	None	Checkpoint triggering interval.
execution.checkpointing.mode	EXACTLY_ONCE	Checkpoint mode. The value can be **EXACTLY_ONCE** or **AT_LEAST_ONCE**.
execution.checkpointing.timeout	10 min	Checkpoint timeout interval.
state.backend	hashmap	State backend type. The value can be **hashmap** or **rocksdb**.

Usage Example

Example 1: Viewing and setting the parallelism degree

 Copyright © Huawei Technologies Co., Ltd.

```
-- Create a DataGen source table.
CREATE TABLE datagen_source (
id BIGINT,
name STRING,
amount DECIMAL(10, 2)
) WITH (
'connector' = 'datagen',
'rows-per-second' = '100',
'fields.id.kind' = 'sequence',
'fields.id.start' = '1',
'fields.id.end' = '10000',
'fields.name.length' = '8'
);
-- Create the Print result table.
CREATE TABLE print_sink (
id BIGINT,
name STRING,
amount DECIMAL(10, 2)
) WITH (
'connector' = 'print',
'print-identifier' = 'set_demo'
);
-- Set the parallelism to 4.
SET 'parallelism.default' = '4';
-- Submit the job.
INSERT INTO print_sink
SELECT id, name, amount
FROM datagen_source;
```

Example 2: Setting the checkpoint and running mode

```
-- Create a DataGen source table.
CREATE TABLE datagen_source (
order_id BIGINT,
customer_id INT,
order_time TIMESTAMP(3),
WATERMARK FOR order_time AS order_time - INTERVAL '5' SECOND
) WITH (
'connector' = 'datagen',
'rows-per-second' = '50',
'fields.order_id.kind' = 'sequence',
'fields.order_id.start' = '1',
'fields.order_id.end' = '50000',
'fields.customer_id.min' = '1',
'fields.customer_id.max' = '1000'
);
-- Create the Print result table.
CREATE TABLE print_sink (
order_id BIGINT,
customer_id INT
) WITH (
'connector' = 'print',
'print-identifier' = 'checkpoint_demo'
);
-- Set the checkpoint interval to 30 seconds.
SET 'execution.checkpointing.interval' = '30s';
-- Set the checkpoint mode to EXACTLY_ONCE.
SET 'execution.checkpointing.mode' = 'EXACTLY_ONCE';
-- Set the state TTL to 1 hour.
SET 'table.exec.state.ttl' = '3600000 ms';
-- Submit the job.
INSERT INTO print_sink
SELECT order_id, customer_id
FROM datagen_source;
```

 Copyright © Huawei Technologies Co., Ltd.

8 Enhanced Flink Features

8.1 Enhanced Functions

8.1.1 BITAND

Syntax

```
INT BITAND(INT number1,INT number2)
```

Description

Performs a bitwise AND operation. The bitwise AND operation compares the binary bits of two integers bit by bit. If both corresponding bits are **1**, the result bit is **1**. Otherwise, the result bit is **0**.

Input Parameters

Parameter	Data Type
number1	INT
number2	INT

NOTE

The input and output parameters must be of the same integer type.

Example

```
CREATE TABLE source (
  f_sequence INT
) WITH (
  'connector' = 'datagen',
  'rows-per-second' = '1'
);
CREATE TABLE print with(
```

 Copyright © Huawei Technologies Co., Ltd.

```
'connector' = 'print'
) as select
  BITAND (2, 3) as int1
from source;
```

The returned result is 2, and the data type is INT.

8.1.2 BITNOT

Syntax

```
INT BITNOT(INT number)
```

Description

The BITNOT function is used to perform a bitwise NOT operation. The bitwise NOT operation inverts the binary bits of an integer, that is, 0 becomes 1 and 1 becomes 0. Because the INT type uses two's complement representation, the result after inversion must be understood according to the two's complement rules.

Input Parameters

Parameter	Data Type
number	INT

Example

```
CREATE TABLE source (
  f_sequence INT
) WITH (
  'connector' = 'datagen',
  'rows-per-second' = '1'
);
CREATE TABLE print with(
'connector' = 'print'
) as select
  BITNOT (7) as int1
from source;
```

The returned result is -8, and the data type is INT.

8.1.3 BITOR

Syntax

```
INT BITOR(INT number1, INT number2)
```

Description

The BITOR function is used to perform the bitwise OR operation. The bitwise OR operation compares the binary bits of two integers bit by bit. If at least one of the two corresponding bits is **1**, the result bit is **1**. Otherwise, the result bit is **0**.

 Copyright © Huawei Technologies Co., Ltd.

Input Parameters

Parameter	Data Type
number1	INT
number2	INT

NOTE

The input and output parameters must be of the same integer type.

Example

```
CREATE TABLE source (
  f_sequence INT
) WITH (
  'connector' = 'datagen',
  'rows-per-second' = '1'
);
CREATE TABLE print with(
'connector' = 'print'
) as select
  BITOR (2, 3) as int1
from source;
```

The returned result is 3, and the data type is INT.

8.1.4 BITXOR

Syntax

```
INT BITXOR(INT number1, INT number2)
```

Description

The BITXOR function is used to perform the bitwise XOR operation. The bitwise XOR operation compares the binary bits of two integers bit by bit. If the two corresponding bits are different, the result bit is **1**. If they are the same, the result bit is **0**.

Input Parameters

Parameter	Data Type
number1	INT
number2	INT

NOTE

The input and output parameters must be of the same integer type.

 Copyright © Huawei Technologies Co., Ltd.

Example

```
CREATE TABLE source (
  f_sequence INT
) WITH (
  'connector' = 'datagen',
  'rows-per-second' = '1'
);
CREATE TABLE print with(
'connector' = 'print'
) as select
  BITXOR (2, 3) as int1
from source;
```

The returned result is 1, and the data type is INT.

8.1.5 DATE_ADD

Syntax

```
STRING DATE_ADD(VARCHAR startdate, INT days)
STRING DATE_ADD(TIMESTAMP time, INT days)
```

Description

The DATE_ADD function is used to return the date after a specified number of days are added to a date. If the target number of days is negative, the corresponding number of days is subtracted.

Parameter Description

- Date: The data type is TIMESTAMP or STRING (format: yyyy-MM-dd HH:mm:ss). The value can be NULL.
- Number of days to add: The data type is INT.
- Return date: The date obtained by adding the specified number of days to the date. The data type is STRING.

Example

```
CREATE TABLE source (
  time1 TIMESTAMP
) WITH (
  'connector' = 'datagen',
  'rows-per-second' = '1'
);
CREATE TABLE Sink (
  date1 string,
  date2 string,
  date3 string
) with ('connector' = 'print');
INSERT INTO
  Sink
SELECT
  DATE_ADD(time1, 30) as date1,
  DATE_ADD('2017-09-15 00:00:00', 30) as date2,
  DATE_ADD(cast(null as timestamp),30) as date3
FROM source
```

 Copyright © Huawei Technologies Co., Ltd.

Table 8-1 Returned Result

date1 (string)	date2 (string)	date3 (string)
2024-06-28	2017-10-15	null

8.1.6 DATE_SUB

Syntax

```
STRING DATE_SUB(STRING startdate, INT days)
STRING DATE_SUB(TIMESTAMP time, INT days)
```

Description

The DATE_SUB function is used to return the date obtained by subtracting a specified number of days from a date. If the target number of days is negative, the corresponding number of days is added.

Parameter Description

- Date: The data type is TIMESTAMP or STRING (format: yyyy-MM-dd HH:mm:ss). The value can be NULL.
- Number of days to add: The data type is INT.
- Return date: The date obtained by subtracting the specified number of days from the date. The data type is STRING.

Example

```
CREATE TABLE source (
  time1 TIMESTAMP
) WITH (
  'connector' = 'datagen',
  'rows-per-second' = '1'
);
CREATE TABLE Sink (
  date1 string,
  date2 string,
  date3 string
) with ('connector' = 'print');
INSERT into
  Sink
select
  DATE_SUB(time1,30) as date1,
  DATE_SUB('2017-09-15 00:00:00', 30) as date2,
  DATE_SUB(cast(null as timestamp),30) as date3
FROM source
```

Table 8-2 Returned Result

date1 (string)	date2 (string)	date3 (string)
2024-04-29	2017-08-16	null

 Copyright © Huawei Technologies Co., Ltd.

8.1.7 DATEDIFF

Syntax

```
INT DATEDIFF(VARCHAR enddate, VARCHAR startdate)
INT DATEDIFF(TIMESTAMP enddate, VARCHAR startdate)
INT DATEDIFF(VARCHAR enddate, TIMESTAMP startdate)
INT DATEDIFF(TIMESTAMP enddate, TIMESTAMP startdate)
```

Description

The DATEDIFF function is used to calculate the number of days between two dates. The calculation subtracts **startdate** from **enddate**, meaning the return value equals **enddate** – **startdate**. A positive number is returned if **enddate** is later than **startdate**, and a negative number is returned if **enddate** is earlier than **startdate**.

Input Parameters

Parameter	Data Type
enddate	TIMESTAMP and VARCHAR
startdate	TIMESTAMP and VARCHAR

NOTE

- VARCHAR date format: yyyy-MM-dd or yyyy-MM-dd HH:mm:ss
- If any input parameter is NULL, NULL is returned.

Example

```
CREATE TABLE source (
  time1 TIMESTAMP
) WITH (
  'connector' = 'datagen',
  'rows-per-second' = '1'
);
CREATE TABLE print with(
'connector' = 'print'
) as select
DATEDIFF('2025-10-15 00:00:00', '2025-09-15 00:00:00') as int1,
       DATEDIFF(time1 ,'2025-12-01 00:00:00') as int2,
       DATEDIFF('2025-12-01',time1 ) as int3,
       DATEDIFF(time1 ,CAST(null as VARCHAR)) as int4,
       DATEDIFF(CAST(null as VARCHAR), '2025-12-01 00:00:00') as int5,
       DATEDIFF(CAST(null as VARCHAR), time1 ) as int6,
       DATEDIFF(time1 ,time1 ) as int7
from source;
```

Table 8-3 Returned Result

int1(INT)	int2(INT)	int3(INT)	int4(INT)	int5(INT)	int6(INT)	int7(INT)
30	49	-49	null	null	null	0

 Copyright © Huawei Technologies Co., Ltd.

8.1.8 URL_DECODE

Syntax

```
VARCHAR URL_DECODE(VARCHAR input)
VARCHAR URL_DECODE(VARCHAR input, VARCHAR encoding)
```

Description

The **URL_DECODE** function is used to convert an input string in **application/x-www-form-urlencoded MIME** format to a standard string.

Input Parameters

Parameter	Data Type	Description
input	VARCHAR	This parameter specifies the input string.
encoding	VARCHAR	This parameter is optional. It specifies the decoding format, which can be GBK or UTF-8. If this parameter is not specified, the default value UTF-8 is used.

Example

```
CREATE TABLE source (
  f_sequence INT
) WITH (
  'connector' = 'datagen',
  'rows-per-second' = '1'
);
CREATE TABLE print with(
'connector' = 'print'
) as select
  URL_DECODE ('http://calcite.apache.org') as value1,
  URL_DECODE ('https%3A%2F%2Fcalcite.apache.org') as value2,
  URL_DECODE ('http%3A%2F%2Ftest%3Fa%3Db%26c%3Dd') as value3,
  URL_DECODE ('http%3A%2F%2F%E4%BD%A0%E5%A5%BD') as value4,
  URL_DECODE ('test') as value5,
  URL_DECODE ('https%%3A%2F%2Fcalcite.apache.org') as value6,
  URL_DECODE ('https%3A%2F%2Fcalcite.apache.org%','UTF-8') as value7,
  URL_DECODE ('https%3A%2F%2Fcalcite.apache.org', 'gb2312') as value8
from source;
```

 Copyright © Huawei Technologies Co., Ltd.

Table 8-4 Returned Result

value1(VARCHAR)	value2(VARCHAR)	value3(VARCHAR)	value4(VARCHAR)	value5(VARCHAR)	value6(VARCHAR)	value7(VARCHAR)	value8(VARCHAR)
http://calcite.apache.org	http://calcite.apache.org	http://test?a=b&c=d	http://Hello	test	null	null	https://calcite.apache.org

8.1.9 URL_ENCODE

Syntax

```
VARCHAR URL_ENCODE(VARCHAR input)
VARCHAR URL_ENCODE(VARCHAR input, VARCHAR encoding)
```

Description

The **URL_ENCODE** function is used to encode an input string in **application/x-www-form-urlencoded MIME** format and return the encoded string.

Input Parameters

Parameter	Data Type	Description
input	VARCHAR	This parameter specifies the input string.
encoding	VARCHAR	This parameter is optional. It specifies the encoding format, which can be GBK or UTF-8. If this parameter is not specified, the default value UTF-8 is used.

Example

```
CREATE TABLE source (
  f_sequence INT
) WITH (
  'connector' = 'datagen',
  'rows-per-second' = '1'
);
CREATE TABLE print with(
'connector' = 'print'
) as select
  URL_ENCODE ('http://calcite.apache.org') as value1,
  URL_ENCODE ('http://test?a=b&c=d') as value2,
  URL_ENCODE ('http://Hello') as value3,
  URL_ENCODE ('test') as value4,
  URL_ENCODE ('http://Hello', 'gb2312') as value5
from source;
```

 Copyright © Huawei Technologies Co., Ltd.

Table 8-5 Returned Result

value1(VARCHAR)	value2(VARCHAR)	value3(VARCHAR)	value4(VARCHAR)	value5(VARCHAR)
http%3A%2F%2Fcalcite.apache.org	http%3A%2F%2Ftest%3Fa%3Db%26c%3Dd	http%3A%2F%2F%E4%BD%A0%E5%A5%BD	test	http%3A%2F%2F%C4%E3%BA%C3

8.2 Using RocksDB Merge Operator in FlinkSQL to Improve the Performance of the Join Operator

Scenario

In FlinkSQL jobs that use the RocksDB state backend, the Join operator with **NoUniqueKey** needs to read the old value and then write the new value when updating the state. This causes a large amount of read amplification overhead, which deteriorates the performance of the Join operator. After this feature is enabled, the Join operator can directly update the state through the merge operation, improving the computing performance of the Join operator.

Notes and Constraints

- This feature works only when the RocksDB state backend is used.
- Snapshots generated before and after this feature is enabled are incompatible.

Usage

You can use either of the following methods to set **table.exec.join.use-merge-operator** to **true** to enable the MergeOperator feature:

- **Method 1**: When configuring a Flink job, add this custom parameter to the **flink_config** parameter of the job.
- **Method 2**: Add this custom parameter to the Flink parameters of the endpoint.

8.3 Using RocksDB Prefix Bloom Filter in FlinkSQL to Improve the Performance of the Join and Top N Operators

Scenario

RocksDB Prefix Bloom Filters can be used to optimize the performance of join and top *N* operators in Flink SQL.

 Copyright © Huawei Technologies Co., Ltd.

Notes and Constraints

Snapshots generated before and after this feature is enabled are incompatible.

Usage

You can use either of the following methods to set **table.exec.use-map-prefix-bloomfilter** to **true** to enable the RocksDB Prefix Bloom Filter feature:

- **Method 1**: When configuring a Flink job, add this custom parameter to the **flink_config** parameter of the job.
- **Method 2**: Add this custom parameter to the Flink parameters of the endpoint.

You can use either of the following methods to set **state.backend.rocksdb.auto-workload-parameters-derivation** to **true** to enable the RocksDB automatic workload parameter derivation feature:

- **Method 1**: When configuring a Flink job, add this custom parameter to the **flink_config** parameter of the job.
- **Method 2**: Add this custom parameter to the Flink parameters of the endpoint.

After the RocksDB automatic load parameter derivation function is enabled, the parameters related to the prefix bloom filter can be automatically adjusted based on the job load.

8.4 Optimizing the Performance of the SPLIT_INDEX Function in Flink SQL

Scenario

When the built-in **SPLIT_INDEX** function of FlinkSQL is used to parse multiple fields of a string, enabling the **SPLIT_INDEX** function performance optimization allows the parsing result of the last split to be reused when parsing multiple fields of the same string, thereby improving computing performance.

Usage

You can use either of the following methods to set **table.optimizing.split-index-optimization** to **true** to enable the **SPLIT_INDEX** function performance optimization:

- **Method 1**: When configuring a Flink job, add this custom parameter to the **flink_config** parameter of the job.
- **Method 2**: Add this custom parameter to the Flink parameters of the endpoint.

SQL Example

```
CREATE TABLE datagen (
  auction BIGINT,
  bidder BIGINT,
```

 Copyright © Huawei Technologies Co., Ltd.

```
  price BIGINT,
  channel VARCHAR,
  url VARCHAR
)
WITH
  ('connector' = 'datagen');

CREATE TABLE blackhole (
  auction BIGINT,
  bidder BIGINT,
  price BIGINT,
  channel VARCHAR,
  dir1 VARCHAR,
  dir2 VARCHAR,
  dir3 VARCHAR
)
WITH
  ('connector' = 'blackhole');

INSERT INTO
  blackhole
SELECT
  auction,
  bidder,
  price,
  channel,
  SPLIT_INDEX (url, '/', 3) as dir1,
  SPLIT_INDEX (url, '/', 4) as dir2,
  SPLIT_INDEX (url, '/', 5) as dir3
FROM
  datagen;
```

8.5 Using Object Reuse of FlinkSQL to Improve Computing Performance

Scenario

In Flink SQL services developed using the Table API, objects are frequently created and destroyed during data processing. This results in significant memory allocation and garbage collection (GC) overhead, which impacts computing performance. After the object reuse feature is enabled, FlinkSQL can reuse data transmission objects between operators, avoiding frequent object creation and destruction, thereby improving SQL computing performance.

Notes and Constraints

- This feature works only for SQL services developed using the Table APIs.
- Objects in Flink SQL UDFs cannot be reused to ensure computing accuracy.

Usage

You can use either of the following methods to set **pipeline.serialization-elimination** to **true** to enable the object reuse function:

- **Method 1**: When configuring a Flink job, add this custom parameter to the **flink_config** parameter of the job.
- **Method 2**: Add this custom parameter to the Flink parameters of the endpoint.

 Copyright © Huawei Technologies Co., Ltd.

8.6 Flink SQL Lookup Join's Support for the PROCTIME() Syntax

Scenario

In the Flink Lookup Join syntax, FOR SYSTEM_TIME AS OF must include the **proc_time** field, for example, **FOR SYSTEM_TIME AS OF o.proc_time**. Now, you can use **PROCTIME()** to replace the **proc_time** field in the table and directly use **FOR SYSTEM_TIME AS OF PROCTIME()**. The **proc_time** field no longer needs to be defined in the table. This is also supported by subqueries.

Usage

When configuring a Flink job, Lookup Join allows direct use of **PROCTIME()** to replace the **proc_time** field in the table.

SQL Example

```
CREATE TABLE kafkasource(
  order_id STRING,
  price DECIMAL(32, 2),
  currency STRING
) WITH (
  'connector' = 'kafka',
  'topic' = 'test_source',
  'properties.bootstrap.servers' = '<service-IP-address-of-Kafka-broker-instance>:<Kafka-port>',
  'properties.group.id' = 'testGroup',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'csv',
  'properties.sasl.kerberos.service.name' = 'kafka',
  'properties.security.protocol' = 'SASL_PLAINTEXT',
  'properties.kerberos.domain.name' = 'hadoop.<system-domain-name>'
);
CREATE TABLE customers (id STRING, name STRING, country STRING) WITH (
  'connector' = 'jdbc',
  'url' = 'jdbc:mysql://mysqlhost:ip/customerdb',
  'table-name' = 'customers',
  'username' = 'xxxx',
  'password' = 'xxxx'
);
CREATE TABLE print (
  order_id STRING,
  price DECIMAL(32, 2),
  name STRING,
  country STRING
) WITH ('connector' = 'print');
insert into
  print
SELECT
  o.order_id,
  o.price,
  c.name,
  c.country
FROM
  kafkasource as o
  JOIN customers FOR SYSTEM_TIME AS OF PROCTIME() AS c ON o.order_id = c.id;
```

 Copyright © Huawei Technologies Co., Ltd.

8.7 Optimizing FlinkSQL JSON_VALUE Performance

Scenario

When the built-in JSON_VALUE function parses multiple fields of a JSON item, the parsing result of the previous JSON item is reused to improve the performance.

Usage

When the built-in **SPLIT_INDEX** function of FlinkSQL is used to parse multiple fields of a string, enabling the **SPLIT_INDEX** function performance optimization allows the parsing result of the last split to be reused when parsing multiple fields of the same string, thereby improving computing performance.

SQL Example

```
CREATE TABLE kafkaSource (
   msgValue STRING
) with (
   'connector' = 'kafka'
   ,'topic' = 'topic-1'
   ,'properties.bootstrap.servers' ='Service IP address of the Kafka Broker instance:Kafka port'
   ,'properties.group.id' = 'group-1'
   ,'value.format' = 'raw'
   ,'scan.startup.mode' = 'group-offsets'
   ,'properties.auto.offset.reset' = 'earliest'
   ,'value.fields-include' = 'EXCEPT_KEY'
);
CREATE TABLE print (
   id STRING,
   name STRING
) with (
     'connector' = 'print'
);
INSERT INTO print
SELECT
   JSON_VAL(msgValue, '$.id'),
   JSON_VAL(msgValue, '$.name')
from kafkaSource;
```

8.8 Limiting Read Rate for Flink SQL Kafka and Upsert-Kafka Connector

Scenario

Traffic limiting is required when Kafka and upsert-kafka connector consume data.

Usage

Add the **subtask.scan.records-per-second.limit** parameter to the created source stream table. This parameter indicates the number of Kafka records consumed in a single partition per second. The overall traffic on the source end is **min(source parallelism * subtask.scan.records-per-second.limit, kafka partitions num * subtask.scan.records-per-second.limit)**.

 Copyright © Huawei Technologies Co., Ltd.

SQL Example

```
CREATE TABLE KafkaSource (
  `user_id` VARCHAR,
  `user_name` VARCHAR,
  `age` INT
) WITH (
  'connector' = 'kafka',
  'topic' = 'test_source',
  'properties.bootstrap.servers' = '<IP address of the Kafka broker instance>:<Kafka port>',
  'properties.group.id' = '<testGroup>',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'csv',
  'subtask.scan.records-per-second.limit' = '1000',
  'properties.sasl.kerberos.service.name' = 'kafka',
  'properties.security.protocol' = 'SASL_PLAINTEXT',
  'properties.kerberos.domain.name' = 'hadoop.<System domain name>'
);
CREATE TABLE KafkaSink(
  `user_id` VARCHAR,
  `user_name` VARCHAR,
  `age` INT
) WITH (
  'connector' = 'kafka',
  'topic' = 'test_sink',
  'properties.bootstrap.servers' = '<IP address of the Kafka broker instance>:<Kafka port>',
  'scan.startup.mode' = 'latest-offset',
  'format' = 'csv',
  'properties.sasl.kerberos.service.name' = 'kafka',
  'properties.security.protocol' = 'SASL_PLAINTEXT',
  'properties.kerberos.domain.name' = 'hadoop.<System domain name>'
);
Insert into
  KafkaSink
select
  *
from
  KafkaSource;
```

8.9 Using Multi-Joins in FlinkSQL

Scenario

In FlinkSQL multi-table join scenarios, multiple join operations are executed one by one by default. Each join operation generates an independent intermediate state. When a large number of tables are joined, intermediate states accumulate significantly, consuming substantial storage resources and degrading job performance. FlinkSQL supports multi-join, which allows multiple join operations to be combined into a single multi-join operator in multi-table join scenarios, effectively reducing intermediate states.

Notes and Constraints

- Multi-joins do not support the Join-to-Live feature.
- Snapshots taken before and after a multi-join operation are incompatible.
- When multi-join is used to join multiple tables, the table join keys must be consistent.

 Copyright © Huawei Technologies Co., Ltd.

Usage

You can use either of the following methods to set **table.optimizer.multi-join.enabled** to **true** to enable the multi-join function:

- **Method 1**: When configuring a Flink job, add this custom parameter to the **flink_config** parameter of the job.
- **Method 2**: Add this custom parameter to the Flink parameters of the endpoint.

SQL Example

```
CREATE TABLE datagen1 (id int, f1 int, PRIMARY KEY (id) NOT ENFORCED)
WITH
  (
    'connector' = 'datagen',
    'rows-per-second' = '1'
  );
CREATE TABLE datagen2 (id int, f2 int, PRIMARY KEY (id) NOT ENFORCED)
WITH
  (
    'connector' = 'datagen',
    'rows-per-second' = '1'
  );
CREATE TABLE datagen3 (id int, f3 int, PRIMARY KEY (id) NOT ENFORCED)
WITH
  (
    'connector' = 'datagen',
    'rows-per-second' = '1'
  );

CREATE TABLE datagen4 (id int, f4 int, PRIMARY KEY (id) NOT ENFORCED)
WITH
  (
    'connector' = 'datagen',
    'rows-per-second' = '1'
  );
CREATE TABLE datagen5 (id int, f5 int, PRIMARY KEY (id) NOT ENFORCED)
WITH
  (
    'connector' = 'datagen',
    'rows-per-second' = '1'
  );
CREATE TABLE print (id int, f1 int, f2 int, f3 int, f4 int, f5 int)
WITH
  ('connector' = 'print');
insert into
  print
select
  datagen1.id,
  datagen1.f1,
  datagen2.f2,
  datagen3.f3,
  datagen4.f4,
  datagen5.f5

from
  datagen1
  left join datagen2 on datagen1.id = datagen2.id
  and datagen2.id > 2
  join datagen3 on datagen1.id = datagen3.id
  and datagen3.id > 3
  right join datagen5 on datagen1.id = datagen5.id
  and datagen5.id > 5
  full outer join datagen4 on datagen1.id = datagen4.id
  and datagen4.id > 4
  and datagen1.id > 1;
```

 Copyright © Huawei Technologies Co., Ltd.

Multi Join Operator Supporting Empty Result Pruning

Scenario

During the running of the Multi Join operator, empty result pruning is supported. When the state corresponding to a table is empty, the traversal of this state is skipped. This reduces invalid recursive scans in the Multi Join operator and improves query performance and resource utilization.

Usage

Set **parameter table.optimizer.multi-join.empty-result-pruning.enabled** to **true** to enable the empty result pruning function.

8.10 Flink on Iceberg Supporting Dimension Table Join

Scenario

Flink on Iceberg supports dimension table joins, allowing you to enrich real-time streaming data with static or slowly changing dimension data stored in Iceberg.

Notes and Constraints

- By default, a lookup join loads all data in each concurrent. It is applicable only to dimension tables with a small amount of data.
- Lookup join keys do not support complex data types, such as **ARRAY**, **BINARY**, **DECIMAL**, **TIME**, **MAP**, or **VARIANT**.
- Cache refresh does not support incremental read. The entire table is scanned during each refresh.
- The Iceberg dimension table supports only **INSERT** and **UPDATE_AFTER** messages.

Parameters

Key	Default Value	Value Range	Description
lookup.refresh.async	false	true/false	Specifies whether to refresh the cache asynchronously. • **true**: The background thread is used for refreshing, and the query is not blocked. • **false**: The query is blocked during synchronous refreshing.
lookup.refresh.interval	1h	Duration (for example, 24 hours)	Interval for refreshing all cached data.

 Copyright © Huawei Technologies Co., Ltd.

Key	Default Value	Value Range	Description
lookup.cache.type	AUTO	AUTO / HEAP / ROCKSDB	Cache backend type. • **AUTO**: The backend is automatically selected based on the table size. • **HEAP**: Caffeine heap memory is used. • **ROCKSDB**: Disk-based.
lookup.cache.auto.bytes-threshold	64 MB	Flink MemorySize (for example, 32 MB)	In AUTO mode, if the table size exceeds this threshold, RocksDB is used. Otherwise, HEAP is used.
lookup.cache.max-table-size	512 MB	Flink MemorySize (for example, 256 MB)	If the size of the Iceberg dimension table exceeds this threshold, the table cannot be loaded.
lookup.rocksdb.path	${java.io.tmpdir}/iceberg-lookup-rocksdb	Valid local directory path.	Data storage path in RocksDB mode.

Usage

Step 1 Create a new Iceberg table, for example, **iceberg_catalog.`default`.iceberg_table**. The following is a code example:

```
CREATE TABLE IF NOT EXISTS iceberg_catalog.`default`.iceberg_table (
  id BIGINT,
  data STRING
) PARTITIONED BY (id)
WITH
  ('format-version' = '2');
```

Step 2 SQL example for dimension table join:

```
CREATE TABLE datagen (id BIGINT)
WITH
  (
    'connector' = 'datagen',
    'rows-per-second' = '1'
  );

CREATE TABLE print
WITH
  ('connector' = 'print') AS
SELECT
  *
FROM
  datagen AS o
```

 Copyright © Huawei Technologies Co., Ltd.

```
 LEFT JOIN iceberg_catalog.`default`.iceberg_table /*+ OPTIONS('lookup.refresh.interval' = '24h') */ FOR
SYSTEM_TIME AS OF PROCTIME () AS d ON o.id = d.id;
```

----End

8.11 Flink on Iceberg Supporting the VARIANT Data Type

Scenario

Flink writes data to Iceberg and supports dynamic and nested semi-structured data.

Notes and Constraints

- Only Iceberg V3 tables are supported.
- Only the Iceberg Parquet format is supported.

Related Functions

```
VARIANT PARSE_JSON(VARCHAR jsonStr [, BOOLEAN allowDuplicateKeys])
VARIANT TRY_PARSE_JSON(VARCHAR jsonStr [, BOOLEAN allowDuplicateKeys])
```

Parameter	Data Type	Description
jsonStr	VARCHAR	JSON string to be parsed. If the value of the **jsonStr** parameter is **NULL**, the return value is **NULL**.
allowDuplicateKeys	BOOLEAN	Optional. The options are as follows: • **true**: Retains the value of the last duplicate key in the JSON string. • **false** (default): Returns **NULL** if duplicate keys exist in the JSON string.

Function	Default Error Handling	Return Value for Invalid JSON	Fault Tolerance
PARSE_JSON	An explicit error is reported.	An exception is thrown by default.	The validity of the input must be strictly verified.
TRY_PARSE_JSON	Fails silently.	**NULL** is returned by default.	Abnormal data streams must be compatible.

 Copyright © Huawei Technologies Co., Ltd.

Usage

Step 1 Create a new Iceberg table, for example, **iceberg_catalog.`default`.iceberg_table**. The following is a code example:

```
CREATE TABLE IF NOT EXISTS iceberg_catalog.`default`.iceberg_table (
  id BIGINT,
  test_data1 VARIANT,
  test_data2 VARIANT
) WITH
  ('format-version' = '3');
```

Step 2 Write data of the VARIANT type. SQL example:

```
CREATE TABLE datagen (id BIGINT)
WITH
  (
    'connector' = 'datagen',
    'rows-per-second' = '1'
  );

INSERT INTO
  iceberg_catalog.`default`.iceberg_table
SELECT
  id,
  PARSE_JSON (
    '{"user":{"name":"Bob","address":{"city":"Shanghai","zip":"200000"}},"tags":["dev","flink"]}'
  ),
  TRY_PARSE_JSON (
    '{"user":{"name":"Bob","address":{"city":"Shanghai","zip":"200000"}},"tags":["dev","flink"]}'
  )
FROM
  datagen;
```

Step 3 Read data of the VARIANT type. SQL example:

```
CREATE TABLE print
WITH
  ('connector' = 'print') AS
SELECT
  id,
  JSON_VALUE (test_data1, '$.user.name'),
  JSON_VALUE (test_data2, '$.tags[0]')
FROM
  iceberg_catalog.`default`.iceberg_table /*+ OPTIONS('streaming'='true', 'monitor-interval'='60s') */;
```

----End

 Copyright © Huawei Technologies Co., Ltd.